Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Fakultät für Informatik und Wirtschaftsinformatik

Bachelor-Thesis

Name:	Michael Galetzka
Thema:	Development and evaluation of a scheduling algorithm for parallel hardware tests at CERN
Arbeitsplatz:	CERN, Genève 23
Referent:	Prof. Dr. Hoffmann
Korreferent:	Prof. Dr.-Ing. Laubenheimer
Abgabetermin:	25.07.2012

Karlsruhe, 25.04.2012

Der Vorsitzende des
Prüfungsausschusses

Prof. Dr. Ditzinger

CERN-THESIS-2012-090
//2012

Except where reference is made in the text of this thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis presented by me for another degree or diploma. No other person's work has been used without due acknowledgement in the main text of the thesis. This thesis has not been submitted for the award of any other degree or diploma in any other tertiary institution.

Place and date:

Signature:

Michael Galetzka

Acknowledgements

First of all, I want to thank my colleague Kajetan Fuchsberger and my supervisor Markus Zerlauth for their great support and the pleasure it was to work with them. I would also like to thank all the other members of the CERN personnel for their kindness, passion and their will to share their vast expertise with me.
Furthermore, I would like to thank my professor, Dr. Dirk W. Hoffmann, for his support and constant feedback.

Most importantly, I would like to thank my girlfriend for supporting me in every way possible and for having the patience to be in a long distance relationship for one year.

Michael Galetzka

Abstract

This thesis aims at describing the problem of scheduling, evaluating different scheduling algorithms and comparing them with each other as well as with the current prototype solution. The implementation of the final solution will be delineated, as will the design considerations that led to it.

The CERN Large Hadron Collider (LHC) has to deal with unprecedented stored energy, both in its particle beams and its superconducting magnet circuits. This energy could result in major equipment damage and downtime if it is not properly extracted from the machine. Before commissioning the machine with the particle beam, several thousands of tests have to be executed, analyzed and tracked to assess the proper functioning of the equipment and protection systems.

These tests access the accelerator's equipment in order to verify the correct behavior of all systems, such as magnets, power converters and interlock controllers. A test could, for example, ramp the magnet to a certain energy level and then provoke an emergency energy extraction by one of the magnet controllers. Each test blocks the tested system for the duration of the test, which can take up to several hours.

For safety reasons, the first tests in a system are always the ones with a lower risk of damaging the equipment, e.g. tests that work with low energy levels in the magnets. As soon as these tests have been analyzed and are considered successful, tests with higher risks are allowed to be executed.

To minimize commissioning time and human error in test execution and result verification, a high degree of automation is essential. To achieve this, a new test tracking and execution framework has been developed at CERN, whose central point is a specialized scheduling strategy that decides which test should be executed when. This strategy has to meet certain requirements. For instance, it needs to consider test dependencies, parallel test execution and environment restrictions including equipment nonconformities or work on the equipment under test. For safety reasons, a high degree of dependability is another principal requirement for this component.

Scheduling tests with constraints is a complex problem, and there are many different algorithms to solve it. The challenge of this thesis is not only to implement one of these algorithms but also to make this implementation easy to use, maintainable and reliable.

Zusammenfassung

Das Ziel dieser Thesis ist es, das Problem des Schedulings zu beschreiben, verschiedene Scheduling-Algorithmen zu evaluieren und diese miteinander sowie mit dem derzeitig benutzten Prototypen zu vergleichen. Die Implementation der ausgesuchten Lösung wird beschrieben, sowie die Design-Überlegungen, die zu ihr geführt haben.

Der CERN Large Hadron Collider (LHC) muss mit einer beispiellosen Menge an Energie umgehen, die sowohl in den Teilchenstrahlen, als auch den supraleitenden Magneten gespeichert ist. Diese Energie könnte zu schwerwiegenden Schäden und langen Ausfallzeiten der Anlagen führen, falls sie nicht richtig aus der Maschine extrahiert wird. Bevor die Maschine mit den Teilchenstrahlen in Betrieb genommen wird, müssen mehrere tausend Tests ausgeführt, analysiert und nachverfolgt werden, um die einwandfreie Funktion der Anlagen und der Sicherheitssystem zu gewährleisten.

Diese Tests greifen auf die Anlagen des Beschleunigers zu um das korrekte Verhalten aller Systeme, wie beispielsweise der Magnete, Leistungswandler oder Sicherheitsschaltungen, zu überprüfen. Ein Test könnte zum Beispiel einen Magneten auf ein bestimmtes Energieniveau bringen und anschließend eine Notfall-Energieextraktion durch einen der Magnetcontroller auslösen. Jeder Test blockiert das getestete System für die Dauer des Tests, was mehrere Stunden dauern kann.

Aus Sicherheitsgründen sind die ersten Tests in einem System immer diejenigen mit einem geringeren Risiko, die Ausrüstung zu beschädigen, wie beispielsweise Tests mit einem sehr geringen Energieniveau der Magnete. Sobald diese Tests analysiert wurden und das Ergebnis erfolgreich ist, dürfen Tests mit einem höheren Risiko ausgeführt werden.

Um die Zeit der Inbetriebnahme zu minimieren und die Möglichkeit menschlicher Fehler bei der Ausführung und Analyse der Tests klein zu halten, ist ein hoher Grad an Automatisierung zwingend notwendig. Hierfür wurde am CERN ein neues Framework für die Ausführung und Nachverfolgung von Tests entwickelt, in dessen Zentrum ein spezialisiertes Scheduling-Verfahren steht, das entscheidet, welche Tests zu welchem Zeitpunkt ausgeführt werden sollen. Dieses Verfahren muss bestimmte Anforderungen erfüllen. Diese umfassen Test-Abhängigkeiten, parallelisierte Ausführung der Tests und Umgebungseinschränkungen, wie Nichtkonformität der Anlagen oder Arbeiten an den zu testenden Systemen. Aus Sicherheitsgründen ist ein hohes Maß an Verlässlichkeit eine weitere unerlässliche Vorgabe an diese Komponente.

Das Planen von Tests unter Beschränkungen ist sehr komplex und es existieren viele verschiedene Ansätze um dieses Problem zu lösen. Die Herausforderung dieser Thesis ist es nicht nur einen dieser Lösungen zu implementieren, sondern diese Implementierung zudem einfach, wartbar und verlässlich zu machen.

Contents

Nomenclature

AI	Artificial Intelligence
CERN	European Organization for Nuclear Research (French: Conseil Européen pour la Recherche Nucléaire)
CSP	Constraint Satisfaction Problem
GPU	Graphics Processing Unit
GUI	Graphical User Interface
JVM	Java Virtual Machine
LHC	Large Hadron Collider (the largest particle accelerator at CERN)
Makespan	The time difference between the start and finish of a scheduling plan
QPS	Quench Protection System - an early warning system to protect superconducting magnets from uncontrolled release of energy
UML	Unified Modeling Language

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This chapter describes the systems whose testing requirements have led to the work described in this thesis, as well as the problems that arise when testing them. The most notable among these systems is the LHC, although the problem of test scheduling applies to other systems, too. The necessity of testing certain systems at CERN will be explained as well as the current test procedures; afterwards an overview over the scheduling problem of the current test architecture will be given.

1.1 The Large Hadron Collider (LHC)

CERN's extensive research in particle physics has led to the construction of several particle accelerators, the largest of which is the LHC. It is the biggest scientific instrument ever built and also the world's most powerful particle accelerator [12].

The LHC is a 27 kilometre long circular accelerator built in a tunnel about 100 meters below ground in the Geneva region. Inside it, two separate particle beams are accelerated nearly to the speed of light.[1] When fully accelerated, the two beams collide with each other at specific experiment points where measurements of the collision are taken.

In order to guide the particle beams, thousands of magnets are deployed around the ring. Almost all of them have to be in a superconducting state in order to be able to create the strong magnetic fields required by the high energy of the beams. To achieve this, the magnets must be kept at a temperature of about $-271\,^{\circ}\mathrm{C}$.

Once a magnet is superconducting, it loses all electric resistance and is able to operate with very high electrical currents. These high stored energies pose a major risk for the magnets and other LHC equipment. If one of the magnets loses its superconducting state while being operational, the energy stored in it could damage not only the magnet itself but the surrounding equipment as well[2].

Every year around Christmas the LHC is shut down for several weeks for maintenance work. During this time, old equipment is being replaced or upgraded and new equipment is added. Not

[1] The beams can achieve up to 0.999999991 times the speed of light at the collision point.

[2] On 19 September 2008, one such incident damaged over 50 superconducting magnets and caused one year down time of the LHC.

only does all of this work have to be coordinated, but the modified equipment also has to be tested in an efficient way. This is the reason why development of a special testing system was initiated at CERN in 2011.

One of the challenges for this system is to coordinate and execute the hardware tests required after a shutdown. Many of these tests can be executed in parallel, although there are many constraints that have to be considered when scheduling them for execution. To keep the time for testing minimal a scheduling algorithm is required that optimizes the execution of these tests.

1.2 The test-framework architecture

The main purpose of the hardware tests is to check the correct functionality of all the LHC circuits, which includes the magnets. There are thousands of magnets installed along the LHC and for every magnet there are several tests to be executed. During a test several electrical signals are recorded to be verified afterwards.[3] The verification of the results is done manually by technical experts or, if possible, automatically.

It is of utmost importance to prevent damage to the magnets during these procedures. Therefore the test-framework has to keep tests from being executed if they do not yet meet all the requirements. On the other hand, if the requirements are met, the test-framework must be able to schedule, execute and track thousands of tests in parallel.

To manage this high amount of work, all of the LHC operators and users need to be able to work and test in parallel. In addition to providing the current status information to all of its users[4], the test-framework has to prevent execution conflicts during testing.

Conflicts would be almost unavoidable if the operators were allowed to start the tests directly on the hardware controllers, because the number of tests is just too large for manual scheduling. That is why the operators only have to select the tests they want to execute, which the test-framework will then schedule and execute. Once a test has been started, the test-framework is also responsible for the result collection. Furthermore it has to store the results in the database and provide the new information to the operators and the analysis components.

A diagram of the general system architecture can be seen in figure 1.1. The typical test workflow would be the following:

1. An operator wants to execute several tests using the control GUI.
2. The test-framework server schedules the tests and informs the hardware control servers[5] which tests should be executed.
3. The first tests are executed and the results are collected by the test-framework server.
4. The next tests are scheduled and executed while the analysis for the finished tests is started.
5. The analysis results are collected and the executed tests are signed by technical experts.

[3] An example of a GUI showing some results can be seen in figure A.1.
[4] A screenshot of the GUI used by the operators can be seen in figure A.2.
[5] The LHC Sequencer is an example for a hardware control server; see [6] for a detailed description.

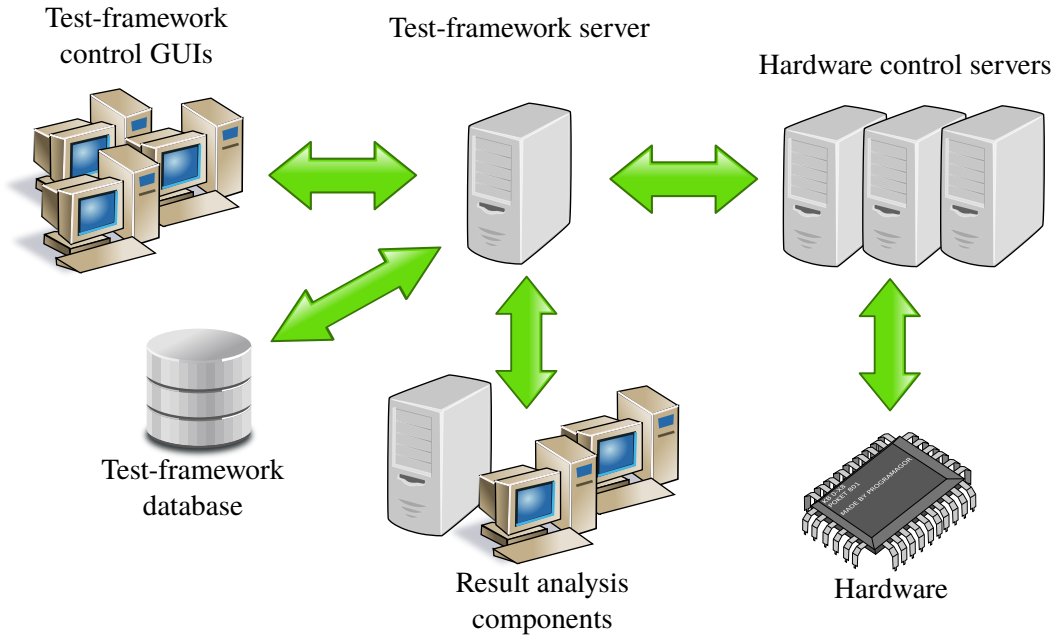


Figure 1.1: Test-framework architecture

This describes just a simplified version of the real workflow. Especially the handling of error states is not included in the example. In any case it is helpful to identify several challenges for the scheduling part of the test-framework.

For one, the test-framework server receives a lot of updates from the GUIs or other systems. In addition, the server itself has to check the status of all running tests on a periodical basis. All of these updates could potentially influence the next scheduling run by allowing or preventing the execution of more tests. This is also the reason why the test scheduling should be very efficient: all pending updates are stalled until the scheduling is finished.

On the other hand, the scheduling strategy has no actual control over the execution and is only able to guess how long a test might take. The scheduling must be very flexible to be able to deal with unexpected test behaviour.

One of the most important points is that the scheduling has to be very reliable. Under no circumstances must a test be started that is not yet allowed to be executed since this could cause serious damage to the hardware.

Some of the tests may influence each other or need the same limited resources. These additional constraints also have to be considered by the scheduling strategy because otherwise the tests might fail or produce wrong results. They would need to be executed again, leading to unnecessary delays.

1.3 The problem of scheduling

Every system has a specific test plan depending on its attributes[6]. This test plan is broken down into test phases which are used to model dependencies between the tests of one particular system.

The rules for these test phases are relatively simple:

- The tests within a phase must not be executed until all the tests of all required test phases have been executed successfully.
- All tests within a phase can be executed in any order.

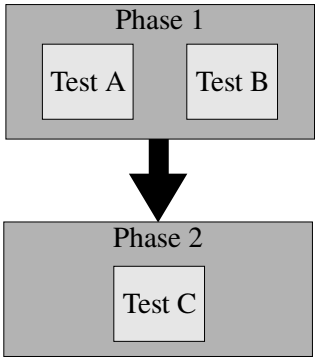


Figure 1.2: An example of two test phases

A small example of a test plan can be seen in figure 1.2. This plan contains two test phases, the first of which, *Phase 1*, has to be successfully executed before *Phase 2* can be executed. This means that *Test C* can be executed if, and only if, *Test A* and *Test B* have been executed and analysed successfully.

If all of the tested systems were independent from each other, scheduling the tests for them would be rather easy. In this example the scheduling strategy would gain nothing from scheduling *Test B* before *Test A*. The only thing to take into account would be to schedule *Test C* last. There would be no way to optimize the maximum time span since it is already optimal. There are no delays between the tests when they are independent from other systems.

In reality, however, the dependencies between the test phases can be complex, as every directed acyclic graph can be used to describe the dependencies in a test plan. It can be very difficult to create a scheduling plan with a minimal makespan when the tests from several systems influence each other.

This becomes obvious already when trying to come up with a scheduling plan for the test plans displayed in figure 1.3. Given the following constraint

> *Test A* must not be executed on more than one system at a time.

What would the best scheduling plan look like?

There are basically two possible scheduling plans, which are depicted in figure 1.4. If all tests have the same duration, the scheduling plan in figure 1.4a will have a 50% longer makespan than the optimal solution.

The problem arises through the introduction of the new constraint and it becomes even more problematic as the number and possibilities of constraints increases. The current test-framework for the LHC has several constraints that have a complex logic when determining if a test is allowed to be executed.

In the end the used constraints originate from the restrictions of the hardware that runs the tests. For the LHC, three kinds of constraints can be identified:

[6] For example the attributes for a magnet might include: location, circuit type, power converter or current level.

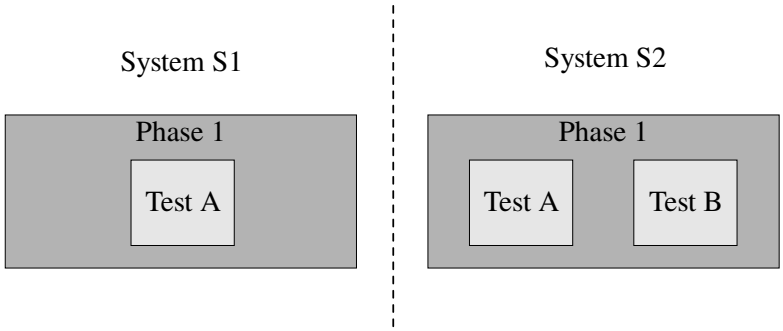
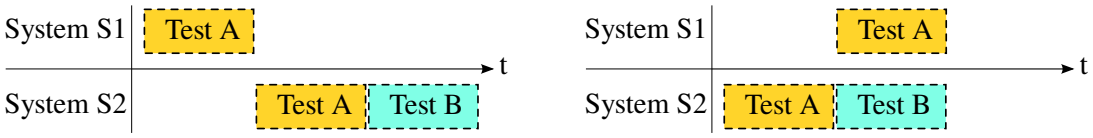


Figure 1.3: Two systems *S1* and *S2* with their test plans



(a) The longer scheduling plan

(b) The optimal scheduling plan

Figure 1.4: Possible scheduling plans for the test plans in figure 1.3

Unary constraints This constraint simply checks restrictions on a single scheduled test. For example, such a constraint could check if some kind of lock is active on the tested system or if a test must not be executed until later that day.

Binary constraint This constraint checks if two given scheduled tests are allowed to be executed in their given configuration. An example would be checking if two tests run at the same time on the same system. Another example would be a constraint checking if two systems are trying to use a limited resource, such as a common QPS controller, at the same time.

Grouping constraint Strictly speaking, this is just another binary constraint, but it is easier to create a scheduling algorithm if this type of constraint is treated separately. The uniqueness of this type of constraint is that it enforces tests to be executed at the same time rather than separately.

For instance, this is the case for some quadrupole magnet pairs inside the LHC. These magnets are located in a combined mechanical structure. If the difference between their currents became too big, the resulting magnetic field would create stress on the equipment. Therefore, it is necessary that a test, which leads to a high current in a magnet, is executed on both magnets concurrently.

So far the only test groups that exist are test pairs, but it is assumed that this will not always be the case. This is another reason why the grouping constraint should be defined in a different way than the binary constraint.

Of course, it is not possible to recreate all restrictions with just these constraints. This can be seen

when looking at the following example: There are three systems which need to execute a certain test X. A restriction exists that prevents the test X to be executed on more than two systems at the same time.

This restriction cannot be represented with a binary constraint because it would require all scheduled tests as input. However, since this more complex type of restriction does not apply in the case of the LHC systems, it can safely be ignored.

Chapter 2

Scheduling

Scheduling problems deal with the allocation of limited resources over time. These resources are used to perform a set of jobs which usually also include some restrictions. Every feasible scheduling plan can be measured by a cost function and it is the aim of a scheduling algorithm to find an allocation with a minimal cost value.

In the case of the LHC hardware commissioning tests, the limited resources are the magnets and power converters, whereas the performed tests are the aforementioned jobs. The algorithm aims to find a scheduling plan for the tests in such a way that the makespan is minimal.

To shorten the description of the algorithms and examples, the following notations are used:

- The expression *[l.n]* used to refer to a single line n of a pseudocode example. Similarly, the expression *[l.n-k]* is used to refer to a range of lines.
- X_{Si} describes the *test* X on the *system* Si. If a test is scheduled more than once for a single system, X^k_{Si} describes the k-th *occurrence* of that test for the system Si. Figure 2.1 displays different example notations.

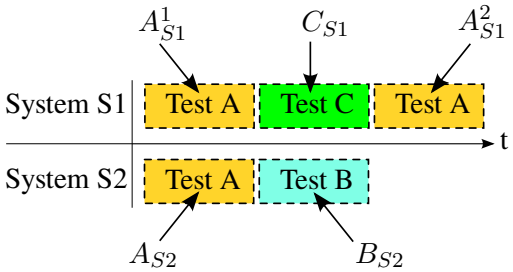


Figure 2.1: Examples of the different test notations.

2.1 The problem of finding an optimal solution

The constraints added to the scheduling make it very hard to find an optimal solution. Finding a set of valid values for these constraints is known as the *constraint satisfaction problem* (CSP) and has been introduced and formalized in the 1960s and 1970s [29, p.51].

The CSP is NP-hard[1]; in other words, no general algorithm that runs in polynomial time and solves the CSP can exist unless P = NP [29, p.28].

Therefore the goal is finding an algorithm that does not always find the optimal solution but that tries to approximate the optimal solution as well as possible. Such algorithms usually work by using some kind of heuristic that exploits information about the problem at hand to narrow down the search space.

Heuristic functions are most widely used in the field of AI, where search is one of the main applications. Although a heuristic search does not perform better than any other search when averaged over all possible cost functions[2], it can still perform much better when the search problem and cost function are well defined.

If defined correctly and applied to the right problem, a good heuristic function guides the search in such a way that it can outperform classic approaches like a tree search. Often, these classic approaches systematically check all possibilities, whereas a heuristic function may skip some solutions or discard some possible search steps. For example, a classic approach to find good moves in a chess match would be to calculate all possible moves for all figures on the board and then chose the ones that will win the match. A heuristic approach might guess a move based on a database that contains previous chess games.

As shown in the example, the use of heuristics usually results in a trade-off between completeness and performance. Although the heuristic approach might choose the wrong move sometimes, it would not take as much time to calculate the next move. However, this trade-off only works if the heuristic is fitted to the problem.

As a general rule, the more information about the problem is put into the used heuristic, the better it will perform. However, one has to be careful not to overfit the heuristic function, otherwise the algorithm can not be efficiently used with similar problems.

For example, it is safe to assume that the scheduling algorithm for hardware tests at the LHC will not be used to schedule thread execution on a microprocessor or to coordinate the assembly of a car. It is, however, very likely that the algorithm might be used to schedule tests for other systems at CERN. It is also safe to assume that the constraints used by the algorithm might change considerably over time and with other applications.

Approximations Since finding the optimal solution for the scheduling problem would take too long, approximation algorithms are the second-best option. After Ronald L.Graham formally introduced them in 1966 to solve a scheduling problem [19], a lot of work has been put into this field of computer science. In addition to classic search techniques, many new approaches, like ant colony optimization, simulated annealing or genetic algorithms, have been developed since then.

[1] For more information about complexity classes like P or NP see [23]

[2] This stems from the *no-free-lunch theorem* by Wolpert and Macready [41].

All of these algorithms share some common characteristics. For one, they are much faster than algorithms that search for an optimal solution. Even though the results might not be optimal, they are often pretty close to optimal. In addition, an approximation factor can be found for most algorithms that describes how much worse the approximated solution can get at most.[3]

Another thing they have in common is that they are not always predictable. They can create *anomalies*, unexpected behaviour that is not known from optimal solutions algorithms. For example, a scheduling algorithm could create a scheduling plan with a smaller makespan if more tests were scheduled. This sounds paradoxical at first, but approximation algorithms often get stuck at local optima and a small change of the input variables can lead to a completely different result.

2.2 The prototype solution

When the new LHC test system was developed in 2011, it had to be ready for the recommissioning of the LHC after the Christmas shutdown. The main goal for the scheduling part was to have a solution that was easy to understand and develop, fast and reliable. Makespan optimizations were not intended and are not even possible since the current algorithm is not able to plan ahead.

All it really does is look at all the tests that need to be scheduled and start as many of them as possible. Its pseudocode is displayed in codeblock 1 and an extended version can be found in codeblock 12 in appendix B.

Algorithm 1 The prototype scheduling solution

```
1:  function SCHEDULEPOSSIBLETESTS
2:      removeImpossibleTests()
3:      pickPossibleTestForEachSystem()
4:
5:      while schedulingPlanHasChanged() do
6:          for all scheduledTests do
7:              if testContradictsGroupConstraint() then
8:                  removeTestFromSchedulingProcess()
9:                  tryToFindReplacementForTest()
10:             end if
11:         end for
12:     end while
13: end function
```

The input given to the algorithm is a list of all the tests the user wants to be executed. The first thing the algorithm does is remove all tests that cannot possibly be scheduled, because they would violate at least one unary constraint *[l.2]*. This includes all the tests for locked systems and

[3] For example the scheduling algorithm described in [19] has an approximation factor of $2 - \frac{1}{m}$, where m is the number of machines available for scheduling. This means the algorithm always performs optimal for one machine and is twice as long as the optimal solution at most when m is increased.

for systems that are already executing tests, as well as the tests that are not allowed to be executed yet because of phase dependencies.

After that, one test is scheduled for each system *[l.3]*. This is done by calling a function for each system that then tries to find a test that does not violate any of the binary constraints.

If there were no grouping constraints, the algorithm would be done at this point. To satisfy the additional constraints, all scheduled tests are checked in a loop until the scheduling plan does not change any more *[l.5-12]*. If one of the tests contradicts a grouping constraint because its partner test has not been scheduled, it is removed from the list of possible tests *[l.8]*. This means that once a test violates a grouping constraint it will not be considered again during that scheduling run, which is not the case if a test violates a binary constraint.

If a replacement can be found for the test that violated the grouping constraint, the replacement is scheduled instead; otherwise no test is scheduled for that system at all *[l.9]*.

Evaluation The presented algorithm has a number of shortcomings. Most notable is its inability to plan ahead, which makes situations like the example in figure 1.4a impossible to prevent. This already happens at the initialization *[l.3]* since the systems are initialized in an arbitrary order. As soon as a test for one system has been chosen, no contradicting test can be picked for a system that is initialized thereafter.

The next problem is that the algorithm has a very limited search space when a grouping constraint has been violated. Tests that violate a grouping constraint will under no circumstance be in the final scheduling plan. This is, of course, a trade-off to speed up the search for a valid plan, but a negative side effect is that it can completely prevent tests from being executed. For example, given the test plans in figure 1.3 and the following grouping constraint:

Test A requires the parallel execution of Test B on another system

what would the resulting scheduling plan be if all three of the tests were to be executed?

First of all, it is assumed that test A will initially be chosen for both the systems $S1$ and $S2$. While checking the grouping constraint, A_{S1} will be removed because B_{S2} is not scheduled. There is no replacement for the removed test as there exists no B_{S1} and subsequently A_{S2} is removed from the scheduling plan. It is replaced with B_{S2}, but since A_{S1} was already removed, the grouping constraint is again violated and B_{S2} is also removed. The result is an empty scheduling plan, although it would be possible to schedule the tests A_{S1} and B_{S2}.

In the presented example, the user who scheduled the tests has no idea why the tests did not start and does not know whether they will be started at all. When taking a look at the source code, one can see that the algorithm tries to give as many hints as possible as to why a certain test could not be started. These comments typically include different constraint violation messages that help the user understand why a test has not yet been started. However, even with the help of these comments, it is almost impossible to predict if or when the tests are going to be executed and what the makespan of the scheduling plan is going to be.

Despite its shortcomings, the algorithm has the big advantage of being very fast compared to other solutions: given n tests to be scheduled, the algorithm requires $\mathcal{O}(n)$ memory and runs in $\mathcal{O}(n^2)$ time.

The linear memory consumption is obvious since the algorithm only needs to keep track of all the n tests and their systems.

The time bound is not as easy to determine. The check of the unary constraints *[l.2]* runs in linear time because a constant number of constraints has to be checked for every test.

The system initialization *[l.3]* has to check the constraints of all the tests that have already been scheduled when trying to find the next possible test for each system. This means that the initialization loop has to check $0 + 1 + .. + (n-2) + (n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$ binary constraints for the n tests.

The check of the grouping constraint *[l.5-12]* normally runs in linear time since there is only a small fixed number of tests that need to be grouped. Even if all tests needed some kind of grouping, the loop would be executed n times at most, because every failed iteration removes a test permanently from the list of possible candidates. At every execution of the for-loop *[l.6-11]*, all scheduled tests have to be checked for constraint violations, which results in a quadratic execution time similar to the initialization loop.

When adding up the worst case execution times from all three parts of the algorithm, the sum has an upper bound of $\mathcal{O}(n^2)$.

2.3 Possible approaches

There are many different approaches to approximate a minimum makespan scheduling plan. All of them are able to find an approximation for the constrained scheduling problem if applied correctly. Of course, the algorithms have different benefits and drawbacks and the aim is to find the algorithm that best fits the requirements given in the introduction chapter.

2.3.1 Generate and Test

A very simple and basic approach to calculate an optimal solution is the *Generate and Test* algorithm. As has been said, this is not an approximation algorithm as it requires an exponential amount of time. It is impractical for real life applications; however, it should be mentioned for the sake of completeness.
The algorithm itself is very simple:

1. generate every possible scheduling plan
2. select the feasible one with the smallest makespan

This is guaranteed to find a solution for a finite CSP [35, p. 118], if one exists, and this solution is guaranteed to be optimal.

2.3.2 Tree Search

The construction of a feasible scheduling plan can be seen as a search in a tree structure. The root node corresponds to an empty plan and every possible addition of a new test creates a new branch in the tree. The leaf nodes correspond to the possible feasible scheduling plans.

In figure 2.2 a simple test plan with two test phases is depicted. The test of the phase P1 must be executed before the tests of the phase P2 can be executed.

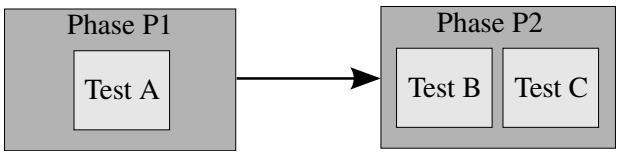


Figure 2.2: A simple test plan with phase dependencies

The search tree for this test plan is shown in figure 2.3. Although the test test plan is rather simple with only three tests and only one system is considered for the schedule, the search tree already consists of 16 different nodes. Of these 16 nodes, only the six leaf nodes are possible solutions. As the number of tests and systems increases, the size of the tree grows exponentially.

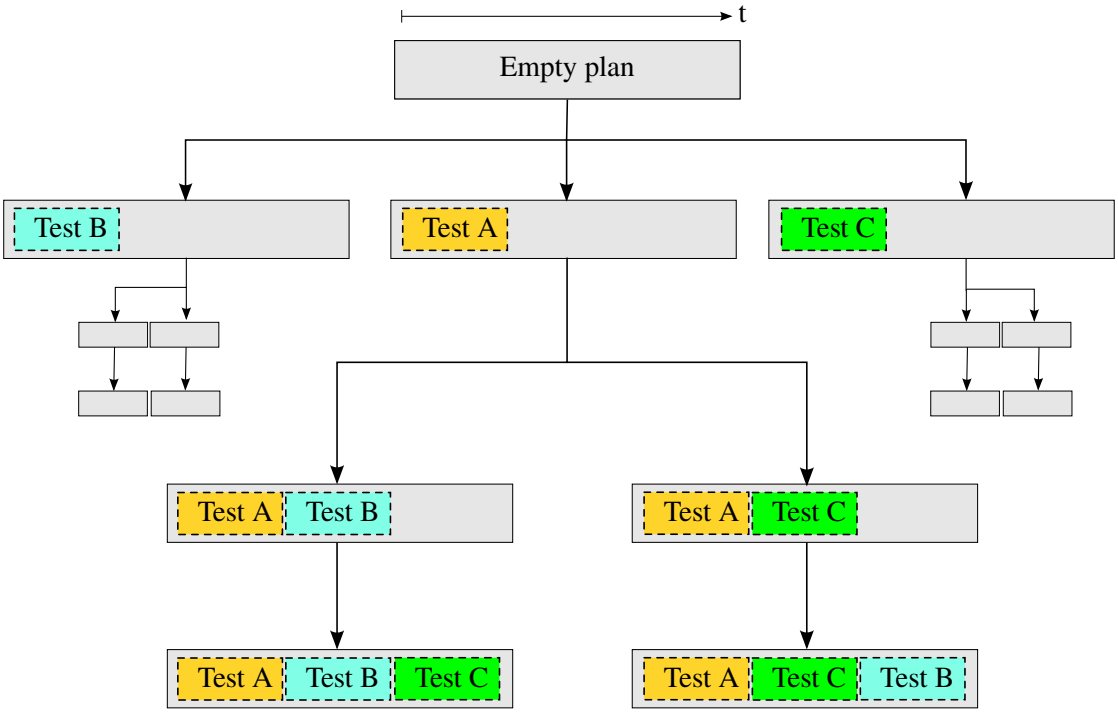


Figure 2.3: The search tree for the test plan given in figure 2.2. Some of the nodes are collapsed to limit the size of the figure.

Not every solution is a feasible one and the search algorithm has to make sure that it does not waste time by traversing paths of the tree that can never lead to a feasible solution. For example, the leftmost path where test B is scheduled first has no feasible solution, because test B can only be scheduled after test A. How the algorithm handles this kind of situation depends on the search order. There are basically two different orders that determine how the tree is traversed.

Breadth-First Search

This search order will expand one layer of the search tree at a time. For example, it would first expand all nodes with a single scheduled test, then all nodes with two scheduled tests and so on. If a node cannot lead to a feasible solution its child-nodes will not be expanded. Once all nodes of a layer have been expanded, it is checked if there are any acceptable solutions among them and the best one of them is selected.

This search technique performs best when the solution is expected very early in the search tree. Due to the fact that all the solutions to the scheduling problem are expected at a depth of k, with k being the number of scheduled tests, the Tree Search is basically equivalent to the *Generate and Test* algorithm with this search order. Even worse, it needs to store the complete tree, causing an exponential memory consumption. This search order can therefore safely be dismissed for the Tree Search.

Depth-First Search

This search order tries to follow a single path as deep as possible and uses backtracking until a solution has been found. When a path to a leaf node is not a feasible solution, the algorithm backtracks to the last point where a feasible solution was still possible and continues the search. The pseudocode for the algorithm is displayed in codeblock 2.

Algorithm 2 Depth-First Search pseudocode

```
1: function DepthFirstSearch(treeNode)
2:     for all child nodes childNode of treeNode do
3:         if isLeafNode(childNode) then
4:             checkSolution(childNode)
5:         else
6:             DepthFirstSearch(childNode)
7:         end if
8:     end for
9: end function
```

The algorithm has the advantage of using very little space, even as the search tree gets very large. The reason is that it just needs to keep track of the current path, which contains at most k elements with k being the depth of the tree. These elements are usually kept on a stack which ensures a simple way of backtracking.

If there are cycles or infinite paths, the search might never stop. However, with a finite search space, like that of the scheduling problem, this search order will always find a solution after some time [17, p.51]. The main problem is that, due to its uninformed search order, the found solution might not be a good approximation. In addition, the performance of this search greatly depends on the ordering of the tree nodes [35, p.81]. For example, if several tests were scheduled for multiple systems, the algorithm could schedule the tests for one system at a time or it could schedule one test per system before scheduling the next ones.

A good heuristic that selects the most promising nodes first can greatly improve the search. However, it is not clear how such a heuristic might look like for the constrained scheduling problem. One approach would be to favour those nodes with the smallest makespan, although this might lead to conflicts more quickly, resulting in more backtracking.

The search can be improved by *pruning*, which is the method of removing sub-trees if they can not have a feasible solution. For example, the pseudocode search method could check all child nodes for constraint violations before recursively calling itself *[l.6]*. If a constraint is violated at this point, the child nodes and all its sub-nodes can be discarded for the search, as there is no way to find a feasible solution with them.

The search can further be improved for a CSP if invalid variable assignments are memorized by the algorithm. This ensures that nodes leading to infeasible paths are not expanded again after a backtrack. The generalization of this technique is called *arc consistency* and can be very helpful when searching in a constraint restricted search space [35, p.120][17, p.575]. This can even be used to apply an *intelligent backtracking* where the algorithm does not jump back a single node, but rather to the node that possibly caused the constraint violation [29, p.360].

Normally, once a solution has been found, the search is finished. In order to get a good approximation it might be necessary to continue the search for a scheduling plan. This requires the definition of a termination criterion, which is not an easy thing to determine. If the criterion is too strict then the search will not be finished within an acceptable time, but if it is too permissive a bad solution might be accepted.

Evaluation The backtracking Depth-First Search technique is the most widely used search technique when trying to find a solution for a CSP. There is a lot of material covering this topic and a lot of research was done on the subject [17, 29, 30, 10, 27].

There exist many variants and optimizations for this search technique. The problem is to find the right technique and apply it to the scheduling problem. Unfortunately, many of the variants, like a look-ahead and look-back search that maintains arc consistency in a constraint network, are rather complex to implement. Although this technique is considered to be the most efficient and complete one to solve a general CSP [29, p.367], other techniques might prove to be a better choice for the scheduling problem.

2.3.3 Ant Colony Optimization

This metaheuristic was inspired by the behaviour of real ants and was first proposed in 1996 to solve hard combinatorial optimization problems [14]. It tries to capture the swarm intelligence of an ant colony in order to solve computational problems. It does so by using artificial ants which imitate the use of pheromones by real life ants.

Every ant in a colony can be seen as an independent, asynchronous agent trying to find a solution for a given problem. For a real ant this might be the search for a food stash; the artificial ants might search for an optimal solution for the presented scheduling problem.

The mechanics of the search is basically the same for both types of ants: each ant tries to find a solution by going through several probabilistic steps and marking these steps with pheromones when a solution has been found. The next ants that are searching for a solution are attracted by

the pheromones and have a higher chance of taking a step marked with many pheromones.

This technique directs the search more and more to the most promising steps and hopefully to the optimal solution. Even when stuck in a local optimum, every ant still has a small chance to find the optimal solution.

In addition to the virtual pheromones, the artificial ants usually also use some kind of problem-specific heuristic to influence their decisions [31]. For example, in the case of the scheduling problem an ant could be less likely to schedule tests with many dependencies at the start of the scheduling plan.

The artificial ants do not use backtracking and might therefore construct infeasible solutions. This might even be allowed to increase the flexibility of the ants when searching for a solution [15, p.8]. The infeasible solution could be penalized depending on its constraint violations to prevent other ant from going the same way.

Algorithm 3 Ant system pseudocode

```
1: function antColonyOptimization
2:     initializeParametersAndPheromones()
3:     while not hasGoodFeasibleSolution() do
4:         letAntsSearchForSolution()
5:         improveSolutionsWithLocalSearch() // Optional
6:         depositPheromones()
7:         performGlobalActions() // Optional
8:     end while
9: end function
```

A simple pseudocode representation for the algorithm is given in codeblock 3. It includes two optional segments:

1. Improving the found solutions with a local search afterwards *[l.5]* is not necessary for the algorithm to work, but it can help to remove constraint violations and find a feasible solution.

2. Global actions *[l.7]* include everything that can not be computed in a distributed way by many ants. For example, the best found solution could gain more weight while the worst found solution could be penalized.

When trying to model the pheromones for the artificial ant there are many possibilities, but one of the simplest is the following: A multidimensional matrix τ stores all the pheromone values whereas a single element τ_{ijk} represents the pheromone value for a system i, a test j and a start time k.

Whenever a single ant found a solution it increases the τ_{ijk} values for all systems and scheduled tests. How much the pheromone values are increased can be influenced by several factors:

- The makespan of the found scheduling plan: the smaller the makespan the more pheromones are deposited.

- Violated constraints reduce the amount of pheromones deposited. The constraints can be separated into soft constraints and hard constraints. A violation of a soft constraint reduces the amount of pheromones deposited, whereas a violation of a hard constraint completely prevents a deposit.

- The ant with the best solution found so far deposits additional pheromones. This is known as *elitist ant* technique and also has a number of disadvantages, as it requires a central point to store the information about the best solution. It is not possible to create a fully distributed ant system with this technique, although the use of separate local elitist ants can solve that problem [40].

Evaluation The algorithm has shown a lot of potential "compared to several other heuristics [...] including genetic algorithms, simulated annealing, tabu search, and different sampling methods [...]" [31].[4]

Since every ant is able to act as an independent agent, the search for a solution can be parallelized very well. It also scales very well with more computing power, as the number of ants can simply be increased to match the hardware. With some adaptations it is even possible to run the ant system on a highly parallel GPU, greatly increasing the speed of the algorithm [11].

Of course, the algorithm also has some drawbacks which stem from its probabilistic nature. For example, it is not possible to predict if and when a good solution will be found. It is also harder to write tests since the algorithm is not deterministic and two consecutive executions with the same parameters can have different results.

It is not clear which additional techniques, like an elitist ant or a distinction between queen and worker ants [11, p.3], are really helpful and when to use them. There are also many parameters, like the amount of deposited pheromones or the number of ants, which can only be determined experimentally. This is due to the fact that the theoretical analysis of the algorithm and its parameters is rather difficult and even the research in this field is experimental rather than theoretical [16, p.5].

It is also not clear at which point the search for a feasible scheduling plan should be stopped. Some termination criterion must be defined that stops the search when a good enough scheduling plan has been found, otherwise the algorithm might run forever. On the other hand, if the criterion is too permissive, the algorithm will also accept bad solutions.

2.3.4 Heuristic Repair

Rather than searching for a feasible scheduling plan by adding single tests until the plan is complete, Heuristic Repair algorithms start with a complete, but infeasible, scheduling plan. This plan is then "repaired" until a feasible solution has been discovered. This repair is usually guided by a heuristic function that provides a good approximation to the optimal solution.

Although not as popular as the Tree Search algorithms, this kind of algorithm has also been extensively researched [32, 33, 37, 38, 39]. The n-queens problem is a classic CSP where this

[4] Although the scheduling problem examined in [31] was not a full CSP, the presented algorithm might be applied for the constrained test scheduling with a few changes.

algorithm has successfully been applied. The aim is to place n queens on a $n * n$ chess boards in such a way that no queen is able to attack any other queen. The Heuristic Repair algorithm is able to solve even large problem instances with 10^6 queens in around 50 steps. [13, p.124] It is practically impossible to find a solution for this problem using a backtrack search algorithm, because the required search tree has a branching factor of 10^{12}.

Although the algorithm performs very well at many large problem instances, like the n-queens problem, it is not suited for every kind of problem. Certain requirements must be met for the algorithm to be successful. For example, the constraints must be permissive enough that there are many possible solutions; this is especially true as the search space gets larger. The constrained scheduling problem satisfies this requirement, because there are many possible scheduling plans that represent feasible solutions, especially as the number of scheduled tests increases.

Algorithm 4 Heuristic Repair pseudocode for the scheduling problem

```
1: function HeuristicRepair
2:     createInitialSchedulingPlan()
3:     while not isFeasibleSolution() do
4:         conflictingTest ← findMostConflictingAssignment()
5:         findNewAssignment(conflictingTest)
6:     end while
7: end function
```

The pseudocode for the Heuristic Repair algorithm is shown in codeblock 4. To decide which test should be reassigned for the repair, the *min-conflicts* heuristic is used: every repair action tries to minimize the number of constraint violations. This is done by finding the test which violates the most constraints and moving it to another assignment.

Normally, the best new assignment is the one where the number of violated constraints is minimal[5], but this is not the best choice when searching for a good scheduling plan. This may not be self-evident at first, but it is due to the fact that the objective of finding a feasible scheduling plan often opposes the objective of finding a scheduling plan with a minimal makespan. For example, given the test plan and start configuration in figure 2.4 and the following constraint, the makespan of the result depends on the approach used to determine the best new assignment:

A_{S2} must not be executed together with C_{S1}.

With the given start configuration, the min-conflict heuristic tries to find a new best assignment for C_{S1} as it violates two constraints, while A^1_{S2} and A^2_{S2} both violate only one constraint. Two different ways for finding the best new assignment are displayed in figure 2.4:

1. The left path displays the approach of using the assignment where C_{S1} violates the *least* constraints. Although this approach is faster than the other one, the result has a longer makespan and is therefore worse.

[5]For example, when solving the n-queens problem, the most conflicting queen is moved to the least conflicting position with every iteration [32].

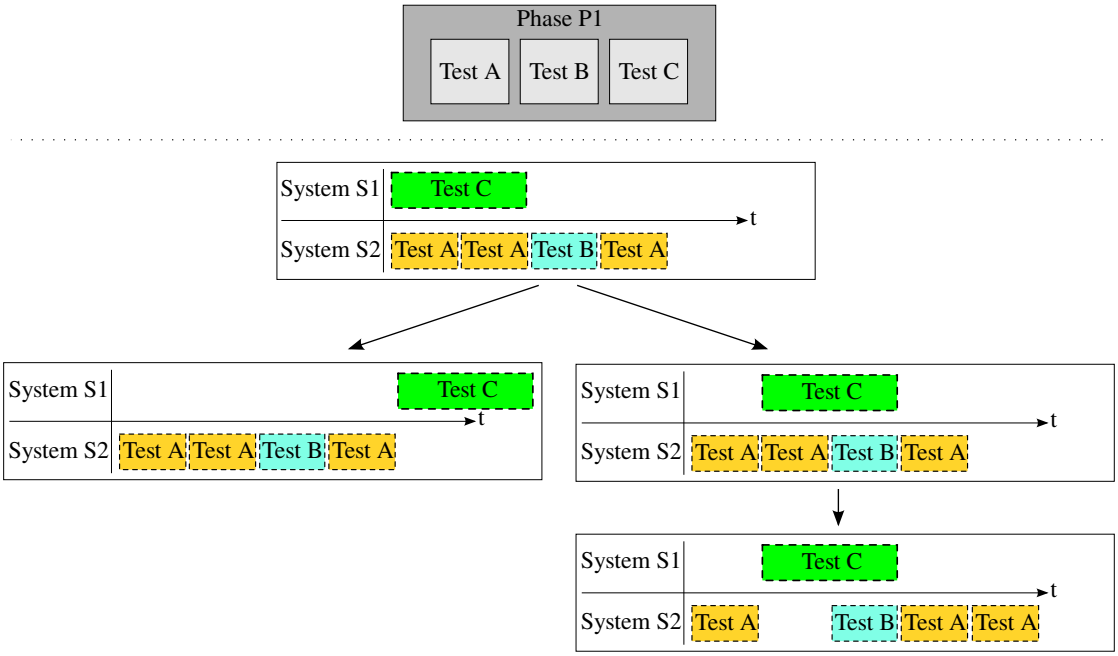


Figure 2.4: A test plan with three phases and a possible start configuration for the Heuristic Repair algorithm. Depending on the approach used to determine new assignments, this leads to different results.

2. The right path displays the second approach, where the best new assignment for C_{S1} depends on two criteria. First, all the assignments that have strictly *less* constraint violations than the original one are searched. From those assignments the one with the smallest makespan is chosen.

   For example, although in the first step the new new assignment of C_{S1} still violates the constraint with A^2_{S2}, it is used because it has one constraint violation less than the original assignment and a smaller makespan than the assignment where C_{S1} violates no constraints at all.

   This approach yields a better result, but takes more steps to complete.

The presented heuristic is a hill-climbing approach, trying to gradually improve the initial state until a solution is found. This has the big disadvantage that the algorithm can get stuck at a local optimum. An example for such a situation is shown in figure 2.5.

When given the following two constraints the algorithm is stuck and can not easily find a solution:

A_{S1} must be executed after B_{S2}.

B_{S2} must be executed after C_{S3}.

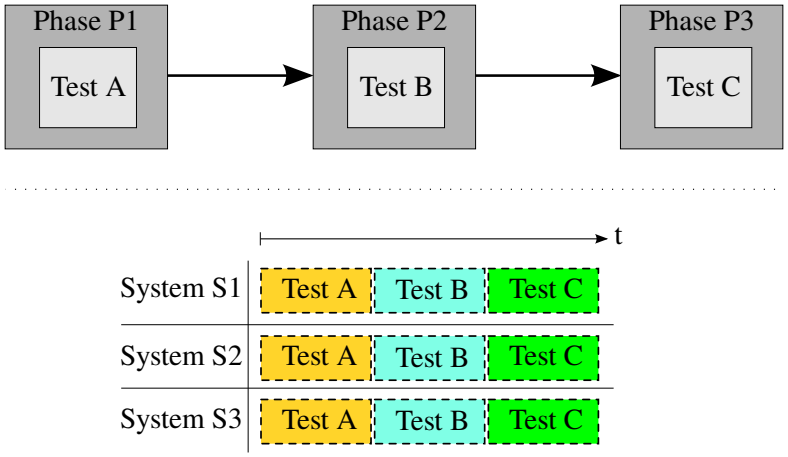


Figure 2.5: A test plan with three phases and a possible start configuration for the Heuristic Repair algorithm. Given certain constraints, the algorithm is stuck at this point.

Clearly, the two constraints are violated right from the start. Using the min-conflicts heuristic, the algorithm tries to reassign B_{S2} as it has two conflicts, whereas A_{S1} and C_{S3} both violate only one constraint. However, if B_{S2} is moved so it starts after C_{S3} ends, it violates against the phase dependencies with C_{S2} and nothing is improved. Likewise, if B_{S2} is moved so it ends before A_{S1} starts, it violates against the phase dependencies with A_{S2} and again, nothing is improved.

Moving A_{S1} or C_{S3} does not improve the situation either, as they would just violate against their phase dependencies.

If the algorithm were to accept a step where the number of violated constraints is not reduced, it could get stuck in an endless loop. It is therefore preferable that, once a local optimum is reached that is not a solution, the algorithm either terminates or finds a way to escape it. However, escaping a local optimum is not always an easy task and has itself been a matter of research [34].

The performance of the algorithm and the quality of the final result are both dependent not only on the used heuristic, but also on the start configuration. While some heuristics work best with a random initialization, other heuristics work better when the solution is already close to the optimal one [38]. Creating a good start configuration is one of the main problems when using a repair based algorithm [32, p.27].

Evaluation Repair based algorithms are seen as "excellent" approach when it comes to solving scheduling problems [32, p.9], because they have a number of advantages:

- They are able to deal with a dynamically changing problem. Tests may be added or removed from the problem, forcing a reschedule. The Heuristic Repair is able to start from the last computed schedule to quickly find a new solution to a changed problem.
- They not only try to find any solution, but rather a good approximation to the optimal one. This is different to other approaches like the Tree Search or Ant Colony Optimization, where just any solution can be found and usually the best of many solutions is chosen.

- They more closely model human behaviour when creating a scheduling plan and it is therefore easier to comprehend how the algorithm came up with the solution [9, p.219].

Compared to other approaches, the algorithm itself is rather easy to implement and understand.

However, the algorithm has the disadvantage that there are many unknowns. For example, it is not clear which start configuration should be used, which heuristic works best for a given problem or how to determine the best new assignment for a constraint violator. Furthermore, it is unclear how to deal with a local optimum that is not a solution.

2.3.5 Genetic Algorithms

This technique was first used in 1975 [22] by simple artificial systems, but nowadays it is used for a great variety of problems. The idea is to mimic the mechanics of biological evolution. Clearly, evolution as stated by Darwin has been very successful, as there are all sorts of life forms which perfectly adapted to their environment. This adaptation enabled them to optimize their gains while reducing their efforts. In the case of a scheduling algorithm the aim is to optimize the quality of the solution while keeping the search effort low.

In analogy to biological evolution, the algorithm works with a number of solution candidates which together form a *population*. Every member of this population encodes his solution information in *chromosomes* which are later used for reproduction. The *fitness* of a member describes how close it is to the optimal solution.

Using an iterative approach, the fittest solution candidates are used to create offspring by reproduction. This reproduction process involves the combination of the parent's *chromosomes* by using genetic operators. These operators usually include some kind of *crossover*, where the chromosomes of two or more parents are combined. Another important genetic operator is *random mutation*, which works by taking a chromosome and randomly changes some of the encoded information. Once the reproduction is finished, the fittest solution candidates are chosen as members of the population for the next iteration.

Algorithm 5 Genetic Algorithm pseudocode for the scheduling problem

```
1: function GeneticAlgorithm
2:     createInitialPopulation()
3:     while not hasGoodFeasibleSolution() do
4:         findCandidatesForReproduction()
5:         reproduceWithCrossoverAndMutation()
6:         generateNewPopulation()
7:     end while
8: end function
```

Codeblock 5 displays the pseudocode for the algorithm. It starts by creating an initial population for the algorithm *[l.2]*. Each member of this population corresponds to a generated scheduling plan. There are different ways to create these initial scheduling plans, but it is sufficient to say that they are created more or less randomly. What is more important is the way the chromosomes of each member are defined, because this influences how effective the genetic operators,

like crossover, are.

Each chromosome consists of certain building-blocks and the *building-block hypothesis*[6] states that a member represents a good solution if it consists of good building-blocks. The initial population is created randomly to obtain a great variety of the search space of different building blocks. It is then the aim of the algorithm to find and combine all the good building-blocks in the population to create a near-optimal solution.

In the case of the scheduling problem, a very simple chromosome could represent the test assignment for a single system and the order of the tests could represent the building-blocks of the chromosome. However, this completely lacks the information about the test dependencies between different systems. It is necessary to use a more complex chromosome description to encode this information, but it is not self-evident how this description should look like.

The iterative part of the algorithm is executed as long as there is no good enough solution *[l.3-7]*. As with the Tree Search and Ant Colony Optimization, it is not clear how to define this stop criterion as it is a trade-off between solution quality and runtime.

Before the reproduction can begin, a number of suitable candidates have to be selected from the current population *[l.4]*. However, it is not easy to determine the best method of selecting these candidates. The selection should not exclusively contain the fittest members of the population, otherwise a single group of highly fit individuals might dominate the selection process and remove the building-blocks of individuals with a low fitness from the algorithm. Therefore several types of parent selection exist, like roulette wheel selection, linear rank selection or tournament selection [2, p.6].

During the crossover part the genetic information of two or more individuals is used to create an offspring by recombining this information. Here also many different methods exist that can be applied, for example order crossover, partially-mapped crossover, cycle crossover [28, p.7], single point crossover, multiple point crossover or uniform crossover [2, p.7]. Which method can be used and works best depends a great deal on other parameters like chromosome design and parent selection.

Figure 2.6 shows the possible results of a simple single point crossover. Using a random crossover point, the offspring is created by arbitrarily taking a part left of the crossover point from one of the parents and then doing the same for the part right of the crossover point. This results in $p^{(c+1)}$ possible offspring combination, where p is the number of parents and c is the number of crossover points. Getting these parameters right is also very important for the performance of the algorithm, but it seems that this has to be done on a trial-and-error basis for each individual problem.

Another step of the reproduction process is the use of the mutation operator. Basically it swaps, adds or deletes a random part of the offspring's genetic information. This helps to escape local optima by creating new starting points for the search and also tries to find new building block which have not been created with the initial population. Of course, the probability of a mutation must not be too large, otherwise the whole search just drifts into chaos and a good solution is almost never found. This is again similar to natural evolution, where some random mutations help to evolve and adapt new features, but too many mutations almost certainly result in death. However, as with the other parameters, there is no real guideline how to optimize the

[6]The hypothesis was first introduced by David Goldberg in [18].

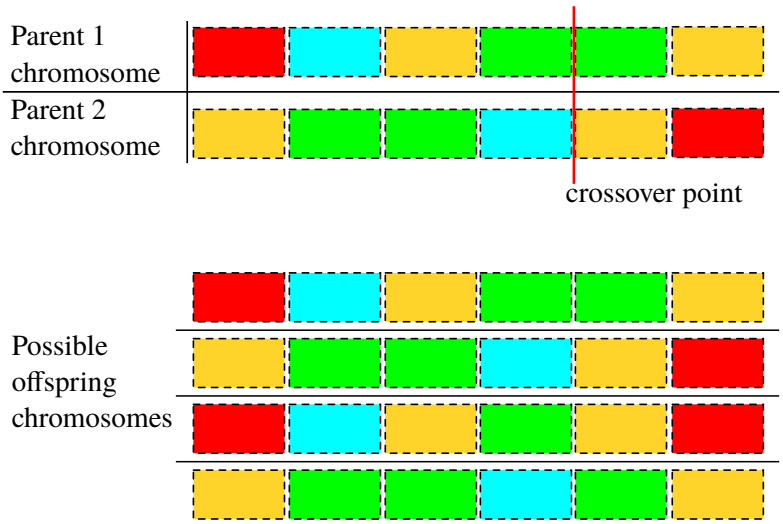


Figure 2.6: A single point chromosome crossover and the possible results.

mutation rate other than through trial-and-error.

After the reproduction has finished, the new population for the next step has to be created. There are several different ways to do that, like selecting all of the offspring or selecting some of the offspring and some individuals from the old population. It is also possible to select the individuals more carefully by using maximum lifespans of individuals or creating tournaments between different generations [2, p.10].

Evaluation The biggest challenge when using a genetic algorithm is to find the right values for all the possible parameters [28, p.8], [36]. For example, one has to experiment with the population size, the stop criterion, the candidate selection for reproduction, the crossover technique and parameters, the genetic mutation, the selection of new population members and some other things. This usually becomes more of a problem for the maintainer of the implemented algorithm, because all of these parameters must be kept track of and eventually changed when the implementation is used for another problem.

On the other hand, Genetic Algorithms are very powerful and have sometimes easily found solutions where other approaches failed [4]. However, this depends on the problem and it is not certain if this good performance can be seen with the scheduling problem.

The algorithm itself is quite easy to understand for anyone who understands the mechanics of the original biological model. Although there are many different variations and different parameters, there is nothing to extraordinary as it is almost always inspired by a natural process. Although the algorithm itself is simple - it is almost impossible to explain why a result was created in a particular way, which is certainly a disadvantage for the users of the algorithm.

2.3.6 Simulated Annealing

This technique was first developed in 1983 and was successfully applied to a large instance of the Travelling Salesman Problem [25]. The main advantage over hill-climbing techniques like Heuristic Repair is the introduction of randomness into the search, which helps to escape local optima. In addition, the algorithm can be implemented in a very generic way, without encoding much information about a specific problem [3, p.920].

The technique tries to simulate the behaviour of matter that is slowly cooled down. For example, the particles in a liquid metal are very mobile and move around each other with a large degree of freedom. When this liquid is cooled down, the particles are more and more bound to their place and form a structure that corresponds to the state of the lowest energy.

Applied to the constrained scheduling problem the algorithm works in the same way. The tests to schedule correspond to the moving particles of the example. They are initialized in a random configuration and moved around until an equilibrium is reached that represents a feasible solution with a makespan that is as small as possible. Every move is randomized which means that it might in fact worsen the solution. However, these moves are allowed with a small probability that gets smaller as the temperature declines.[7] It is this Monte Carlo behaviour that enables the algorithm to escape local optima.

Algorithm 6 Simulated Annealing pseudocode for the scheduling problem

```
1:  function AnnealingAlgorithm
2:      createRandomInitialConfiguration()
3:      setInitialTemperature()
4:      while not hasGoodFeasibleSolution() do
5:          while hasMovesLeftForThisTemperature() do
6:              moveTestToRandomPosition()
7:              if isNewConfigurationBetter() ∨ isAcceptingAlthoughWorse() then
8:                  acceptNewConfiguration()
9:              end if
10:         end while
11:         decreaseTemperature()
12:     end while
13: end function
```

Codeblock 6 shows the pseudocode for the Simulated Annealing algorithm. After the initialization process *[l.2-3]*, the function loops until a good enough feasible solution has been found *[l.4-12]*. As with the other algorithms, like Tree Search or Genetic Algorithms, it is not easy to determine what the criteria for a good solution are and when the algorithm can be terminated.

During the loop, a certain number of tests are moved *[l.5-10]* and after that the simulated temperature is decreased by a factor bigger than one[8] *[l.11]*.

[7] This means that for a temperature $T = \infty$ all the moves are accepted and the outcome is completely random, whereas for $T = 0$ the algorithm becomes greedy and only allows moves that improve the solution.

[8] Usually the value for a temperature T is determined by using the formula $T_i = T_{i-1} * \alpha$ with $0.8 \leq \alpha < 1$ [1, p.114]

Both the number of tests moved during each iteration as well as the exact factor used to decrease the temperature have to be adjusted to the scheduling problem. When trying to determine these values, a trade-off has to be made between quality of the solution and performance of the algorithm. Although the quality of the solution gets better when the number of moved tests per iteration is increased, the required runtime obviously increases [3, p.925]. The same is true if the temperature is decreased at a slower rate. Although there are some guidelines which values to use, they still have to be obtained by testing different possibilities, which can be very time-consuming.

Whenever a test is moved, the new position is determined at random *[l.6]*. This means that a test could violate one or more constraints or increase the overall makespan after it was moved. If this happens, the moved test has to pass a probability check to be accepted, otherwise the movement is discarded *[l.7]*. This check is done in the following way:

1. The worsening of the solution by constraint violations and increased makespan is expressed in a positive number W. The value W increases as the solution gets worse.

   Determining how the problem-specific values like makespan and constraints translate into a single value is not self-evident and has to be evaluated through testing different possibilities.

2. A random value R with $0 < R < 1$ is created.

3. The new position of the moved test if accepted if and only if $e^{-W/T} > R$, where T represents the current simulated temperature of the system [5, p.172]. Clearly, the chance for a move to be accepted decreases as the temperature decreases, as the term $-W/T$ gets bigger. Likewise, the chance to accept a move decreases as the value of W increases.

Although the algorithm itself is not very problem-specific, certain details have to be clarified before it can be applied to the constraint scheduling problem. There exist four basic prerequisites that have to be fulfilled in order to use the algorithm [1, p.116]:

Concise problem representation: This is a requirement for every algorithm, as the elements of the scheduling problem must be represented in a way that is compatible with the algorithm. Especially the representation of the various constraints and their translation into a value function is of great importance.

Neighborhood function: The neighborhood function defines how the result of a single movement operation looks like. Naturally, a single movement operation moves a single test from its current position to any other random position. However, as the topology of this function has a great effect on the effectiveness of the algorithm [21, p.305], other possibilities have to be considered, like e.g. moving several dependent tests at once.

Transition mechanism: This mechanism describes how and under which conditions the state of the system changes. This corresponds to *[l.6-9]* of the pseudocode representation and has already been discussed.

Cooling schedule: The schedule is determined by the rate of the temperature decrease and the number of moved tests at each temperature. It is necessary to find a good balance between the runtime and the quality of the solution.

Although it might take a lot of effort to specify all of these prerequisites, the final implementation then has very few parameters that need to be adjusted when the algorithm is used for other problems. Even if the algorithm is applied to new problems without adjusting them, the worst thing that can happen is that the algorithm takes longer to execute or the solution is not as good as expected.

Evaluation Given a sufficient amount of time, the algorithm has been proven to converge to the best possible solution [3, p.927]. The biggest drawback is that the algorithm usually requires a long time to come up with acceptable solutions. There have been several approaches to optimize the algorithm. Some of them include using heuristics to guide the otherwise random steps [20], other approaches try to parallelize some steps of the algorithm [7], [24].

However, this usually means that the algorithm becomes increasingly complex. For example, it cannot be parallelized without making all the steps independent from each other and defining the workloads for the used processors. These adaptions either cause the algorithm to lose its convergence properties or make it application dependent [3, p.929], which causes it to lose one of its main advantages.

The biggest disadvantage, however, is that studies have shown that the algorithm is not effective when used to solve certain scheduling problems [26]. The algorithm then no longer converges to the optimal solution, but to a local optimum. It has to be combined with other approached, like Genetic Algorithms, to prevent getting stuck. However, this would immensely complicate the implementation, as well as the work needed to maintain it. It is not clear if the constrained scheduling problem of the LHC leads to the same result as the problem described in the studies, but the two problems are similar enough to assume that this is the case.

2.4 Design decision

There is not enough time to implement a prototype for all of the algorithms and compare their runtime performance, during the course of this thesis. Naturally, it is almost impossible to predict which algorithm will have the best performance when applied to the constrained scheduling problem, because in the end this is dependent on many factors like the implementation and the used parameters, the hardware it runs on and the problem size.

However, almost all of the algorithms can be safely dismissed as a possibility for reasons other than the expected performance. Most approaches simply have an implementation complexity that is too high to make them good choice. A more complex solution is harder to implement, maintain and test. As long as the final implementation is able to find a feasible solution within a reasonable amount of time, it should be as simple as possible.

The following list states the dismissed approaches, along with the reasons that led to the decision not to use them:

Tree Search is probably the algorithm that can have the simplest implementation. However, without using one of the many variants and optimizations, the implementation does not perform well enough. Trying to find out which optimizations work best requires a lot of time and adds unnecessary complexity to the implementation. Using a constraint network

to guide the search of the algorithm adds a lot of complexity to the implementation, but it is absolutely necessary when trying to solve a CSP. Therefore, this solution would take too long to implement and be too complex to be effectively maintained.

Ant Colony Optimization has the problem that there are many different parameters used by the implementation. Adjusting them is done by trial and error, which is very time consuming and will likely not be done by the people using and maintaining the implementation. Another problem is that the stochastic nature of the algorithm makes testing very hard, as failed test cases might not be repeated. Therefore, this solution is error-prone and not easily maintainable.

Genetic Algorithms have the same problems as the Ant Colony Optimization algorithm. It uses even more parameters and is also hard to test because of its stochastic nature. In addition, there are many different variants of the algorithm and finding the one that works best for the scheduling problem is not an easy task.

Simulated Annealing has shown to be a very reliable algorithm that, once implemented, does not need much work to be maintained. However, it generally does not have a good performance and the optimization methods require a lot of effort. Nevertheless, the biggest problem is that there is a chance that the algorithm does not work as intended when used for constrained scheduling problems. This renders it practically useless, as it would not only be slower than other approaches, but also create worse results.

Heuristic Repair seems like the best approach possible for several reasons. Even though the best start configuration and heuristic function have to be determined, the final implementation has no parameters left to maintain. The algorithm itself is easy to understand and does not have too many variants. In addition, it is deterministic and easy to test. The operations of the algorithm are easily comprehensible, which is very helpful when writing test cases. Last but not least, the approach showed very good runtime performances when applied to similar problems.

Following the above reasons, the Heuristic Repair algorithm was chosen to be implemented as a solution to the constrained scheduling problem.

Chapter 3

Requirements and Design

The aim of this chapter is to show how the application design was created from the requirements and how this resulted in the final implementation.

3.1 Requirements

Some of the requirements, like good maintainability, have already been stated in the previous chapters. Basically, all of the requirements can be deducted from the implementation's use-cases.

1. The first and most important use-case is its use by the LHC hardware commissioning software. There it has to schedule the different tests for the magnets.

2. The second use-case is that the functionality can be used by applications other than the hardware commissioning software. For example, it could be used to schedule the accelerator beam commissioning or to manage the LHC tunnel access. These applications do not have to be related to the first use case at all, therefore the implementation must be generic enough to allow other applications to use it.

To be used by the hardware commissioning software, the implementation must meet at least the following requirements:

Correctness It is very important that the resulting scheduling plan does not contain any constraint violations. The result created by the scheduling algorithm is directly used by the execution part of the hardware commissioning software. Therefore, wrong scheduling results can lead to unwanted consequences like aborted tests, activation of hardware fail-safe mechanisms or even damaged hardware. Hence, a good and thorough testing of the implementation is mandatory.

Effectiveness If the implementation is not able to produce scheduling plans that are better than the result of the current prototype solution, then it is almost useless. Furthermore, the algorithm should at least be able to produce *some* result, as long as the constraints do not contradict each other. Although this might sound like an obvious requirement, it is not an

easy thing to guarantee, because the algorithm tends to get stuck at local optima that do not qualify as results.

If possible, it would be nice if some additional features could be implemented. For example, the user might want to manually manipulate the scheduling plan by trying to move one or more tests around.

Efficiency The implementation must be able to handle the task of scheduling thousands of tests using the hardware currently provided. This task should ideally not take longer than a few seconds, as a rescheduling might happen quite frequently. Also, the longer the scheduling runs, the longer the start of each test is delayed. Even if the first execution with thousands of tests takes up to a minute to complete, small changes that cause a rescheduling should be finished within seconds. For example, if a test has failed or the user adds a few more tests to be scheduled, the implementation should reuse the existing solution to save time.

Maintainability Due to the ever-changing demands of the hardware commissioning campaigns, the implementation will need to be changed and adapted very often. The parts of the algorithm that change very often, like the scheduling constraints, must be easily accessible and modifiable without a lot of effort. The algorithm implementation should be clearly structured and unusual parts explained, if necessary.

The implementation should not be dependent on the hardware commissioning software by having any references to it. The scheduler implementation should be a separate project that provides a generic scheduling functionality, which can be used by other projects.

3.2 Design details

This section describes the various design decisions from different parts of the implementation.

3.2.1 Data Model

The data model must be able to represent the tests to schedule. On the other hand, it should not have any direct references to tests or magnets. The diagram in figure 3.1 shows the three basic data entities used by the implementation:

1. The host of the scheduled item, like a magnet or any other system, is represented by the class `Lane`. A lane has a number that identifies it and carries no other information. The mapping from a lane to the actual system must be done by the user of the implementation.

2. The items that should be scheduled, like the tests for the magnets, are represented by the class `ItemToSchedule`. It is important to note that in the case of the magnet tests, this class does not just represent a single test, but rather a group of tests that must be executed together because of grouping constraints[1].

[1] The grouping constraints are described in chapter 1.3 on page 12.

So, apart from its id, it stores the duration of each test for each lane. For example, two independent tests A_{S1} and B_{S2} would be represented by two different instances of `ItemToSchedule`. However, if a grouping constraint enforces the tests A_{S1} and B_{S2} to be executed together, they will be represented by a single instance of `ItemToSchedule`.

As with the lanes, the mapping to the actual tests must be made by the user of the implementation. This is possible despite the grouping of several tests, because each group can contain at most one test per system. It is also easily possible for the user of the implementation to simply extend the class `ItemToSchedule` to encode this mapping information.

3. The class `ItemToSchedule` does not contain any information about the actual time its elements are scheduled. The class `ScheduledItem` does exactly that by assigning a start time to a single instance of `ItemToSchedule`.

   For example, the result of the algorithm basically just consists of a collection of `ItemToSchedule` instances. All of the instances with a start value of zero represent tests that should be executed now, while the other instances represent tests that should be executed later.

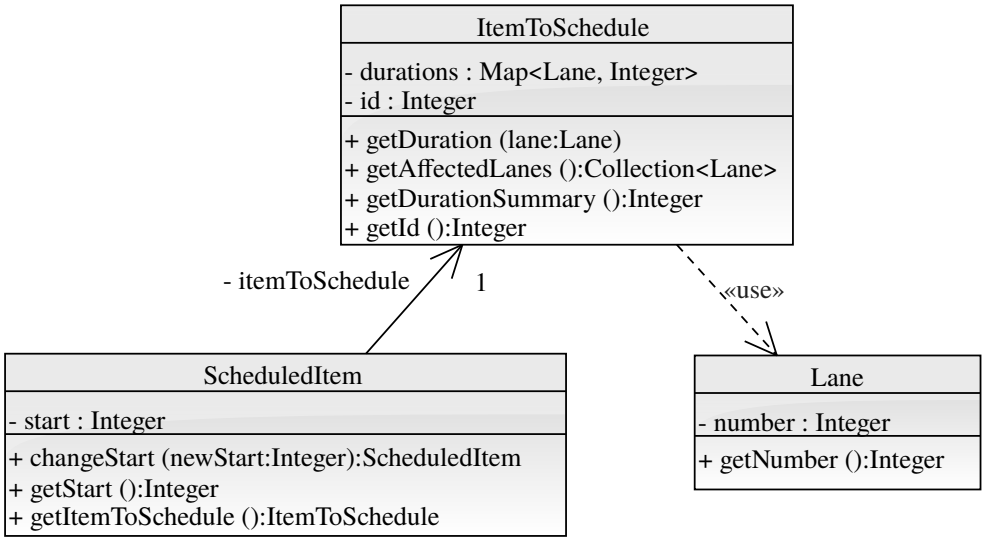


Figure 3.1: Data model diagram.

3.2.2 Constraints

The algorithms works by finding the most conflicting assignment and then finding a new assignment for the conflicting element. The constraints are used to determine when an assignment is conflicting, but this alone is not sufficient for the algorithm. They also have to quantify each conflict, so it can be compared to other conflicts.

The simplest way of calculating the conflict weight is to just use the duration of the scheduled items that are conflicting. Graduations can be achieved by using the overlapping value of the scheduled items.

An example of this is shown in figure 3.2. There are six items scheduled on three different lanes. Additionally, the following two constraint are defined for this example:

Items on the same lane must not overlap.

Item B must be executed after item A.

Although the figure just states the summary violation value, the violation values are determined for each item individually. The assignments on the first lane obviously do not violate any of the constraints and therefore the violation value is zero. The second lane assignments violate the second constraint, therefore item B and item A both get a violation value that corresponds to their duration of 100. The assignments on the third lane violate against both constraints, because Item B is scheduled to start in the middle of item A. So, each scheduled item not only gets a violation value of 100 from the second violated constraint, but also an additional 40 as a result of the overlapping.

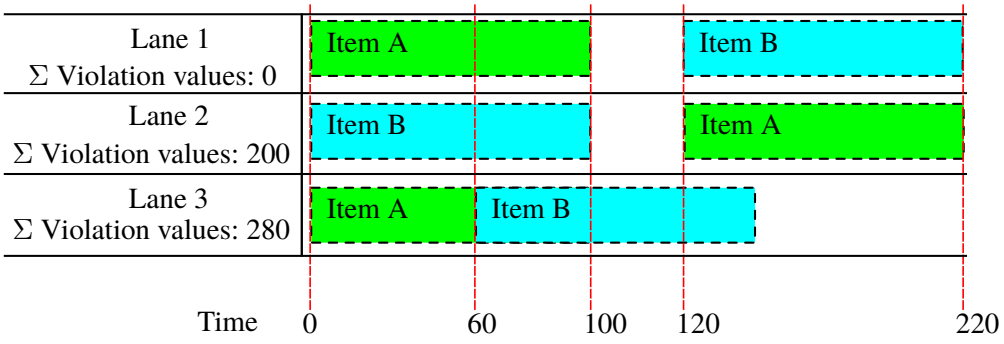


Figure 3.2: An example how the values for constraint violations are determined. This example works with two constraints: 1. Items on the same lane must not overlap; 2. Item B depends on item A.

The constraints shown here are considered *hard* constraints, because if one of them returns a value greater than zero, the assignment is not a valid result. However, if only hard constraints are considered by the algorithm, then gaps between the assignments will not be closed. For example, the first lane in figure 3.2 does not violate any constraints, but there is clearly a gap between the two scheduled items. It is also not easily possible to just add another hard constraint that enforces all gaps to be closed, because sometimes they are indeed necessary.

To solve this problem, the algorithm also uses another type of constraint. These *soft* constraints work exactly like the hard ones, except that the algorithm gives them a lower priority and can finish even if they are violated. For example, a simple soft constraint could be the following: all scheduled items must start at the time 0. It might come as a surprise, but this constraint solves

the problem of gaps between scheduled items. This works, because the constraint tries to move all items as close as possible to the start location of zero. However, as it is only a soft constraint, it must not violate any hard constraint while doing so.

Using the original gap example in figure 3.2, item B would violate the soft constraint with a value of 120, as this is its distance from the start of zero. The only possible new start assignment would be at 100, because any other position either violates a hard constraint or further increases the violation value of the soft constraint.

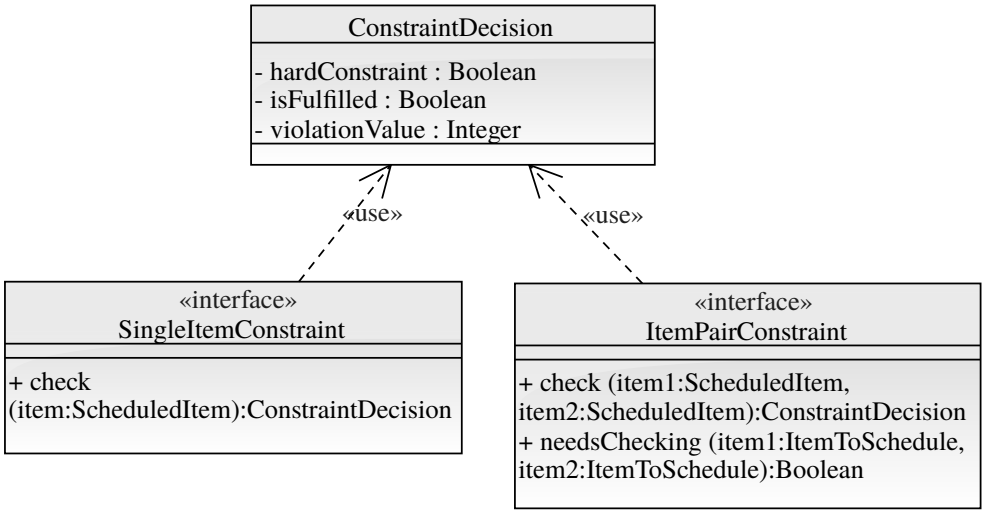


Figure 3.3: Constraint diagram.

Figure 3.3 shows the class diagram for the constraint related classes. The simple unary constraints are represented by the class `SingleItemConstraint`, whereas the binary constraints are represented by the class `ItemPairConstraint`. Both classes define a method to check if the constraint is violated by the specified scheduled items. As a result they return an instance of the class `ConstraintDecision`, that not only states if the constraint was violated, but also if it is a hard or soft constraint, as well as the violation value.

Additionally, the class `ItemPairConstraint` defines a method that determines, if two items need to be checked by this constraint at all. For example, a constraint that checks if two items overlap will only return *true* if the items share a lane. This is very useful, because not only does it speed up the algorithm by skipping unnecessary checks, but it can also be used to find clusters of items that interact with each other. The algorithm could then be executed on each cluster in parallel and the results could then be combined to create the final scheduling plan.

3.2.3 Data Management

The data generated and used by the algorithm, like the scheduled items and constraint violations, is managed by separate classes. For one, all of the scheduled items are managed by the class `SchedulePlan`, which is displayed in figure 3.4. It contains many methods to manipulate and

query the scheduled items. For example, it automatically keeps track of things like the makespan or start values for new scheduled items.

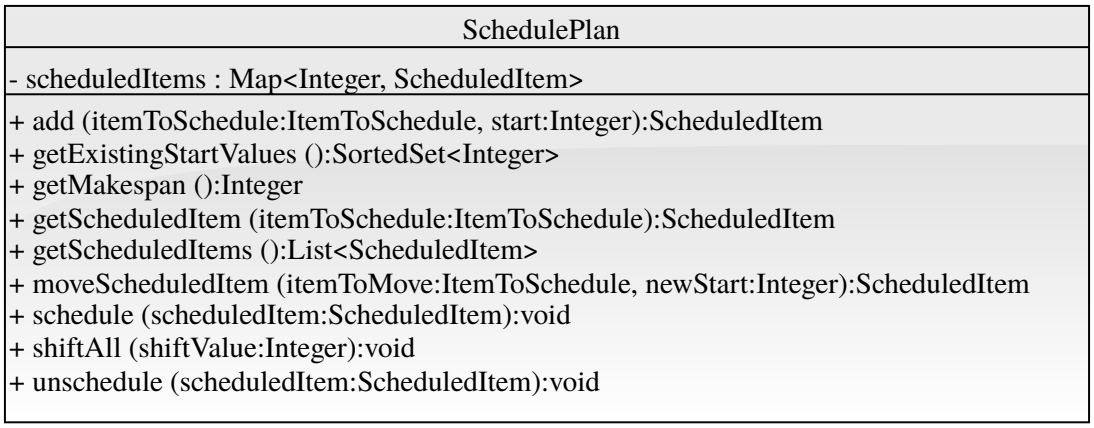


Figure 3.4: Scheduling plan diagram.

Another important task is the management of the constraint violations, because it is critical to the performance of the algorithm. For one thing, the algorithm requires the assignment with the most violated constraints at every iteration. It would take too much time to recompute all of the constraint violations at every iteration. Therefore, the class `ViolationsManager` that handles all of the constraints, was designed to be as efficient as possible. Its class diagram is shown in figure 3.5, along with the data structures it uses.

It stores all of the constraints and uses them to keep track of the violations in a certain scheduling plan. To do this, it first has to be initialized with a complete scheduling plan. This initialization process requires $\mathcal{O}(n^2)$ time, because every scheduled item has to be checked against every other scheduled item. During this initialization, the violations manager creates a number of data structures that enable it to perform subsequent actions very efficiently.

For every scheduled item, all violations are summarized and the result is stored in an instance of the class `Violator`. All of these violators are then kept in a sorted set that allows very fast manipulations in $\mathcal{O}(logn)$ time and returns the biggest element in constant time. However, this alone would be of no use if a small scheduling plan change were to cause a complete recalculation of all the violators.

Therefore, the violations manager uses another data structure to keep track which violators need to be recalculated after a change. The working of this *constraint map* is displayed in figure 3.6. The example in the figure assumes that there exists four items A,B,C and D and that there are constraints between the following items:

- A and B
- A and D
- C and D

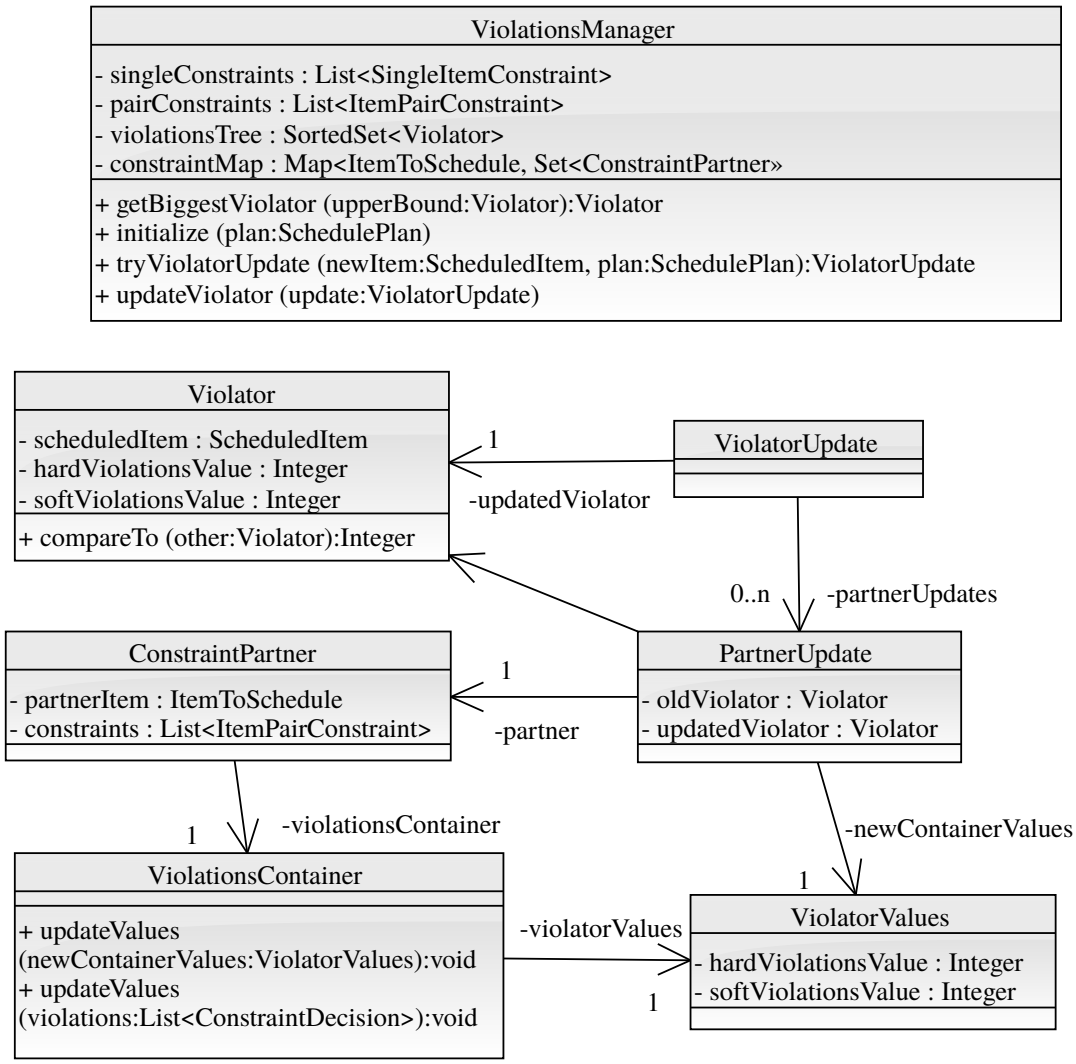


Figure 3.5: Violations manager diagram.

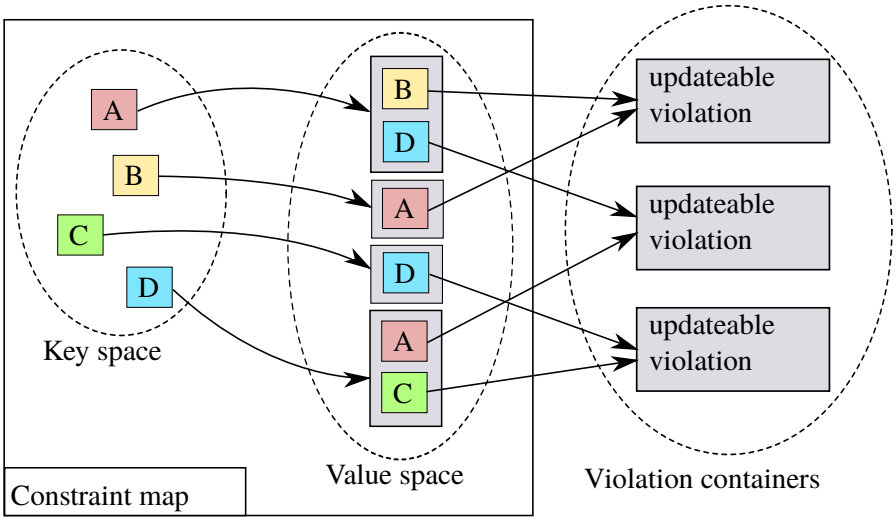


Figure 3.6: The constraint map used to keep track of the constraint violations. There exist constraints between the following items: A and B, A and D, C and D.

The information about the constraints can be obtained by using the `needsChecking`-method of the `ItemPairConstraint` class for every item pair combination possible. So, although the initial constraint check requires $\mathcal{O}(n^2)$ time, any further operations on an item only require to recalculate the violations for its connected items in the map. For example, if the scheduled item D was moved then only the violations for the pairs A-D and C-D would have to be recalculated. This can be seen as the map contains a key D that points to a value containing representatives for the items A and C.

The values of the constraint map are represented by the `ConstraintPartner` class. They do not hold the values of the constraint violations directly because of two reasons:

1. Each constraint violation is accessible in two ways, because there are always two items involved in a violation. For example, the constraint violation of the items A and B is accessible by using item A as key in the constraint map or by using item B as a key. Were the values of the constraint map to store the constraint violations directly then they would need about twice the space for the violations.

2. If each constraint map value held its own version of the constraint violations then an update of one value would lead to an inconsistent state. Since the values of the constraint map do not propagate changes in any way, a changed constraint violation would need two separate value updates.

To prevent these two problems, every constraint-pair of items gets a separate container for their constraint violations. Every constraint violation is saved only once, so that all value entries of the constraint map can see changes at the same time. The class `ViolationsContainer` represents this container. As the constraint map itself, the container is generated only during the

initialization procedure. To prevent updating the constraint map values with new containers every time they are updated, they stay the same and only change their content represented by an instance of the `ViolatorValues` class.

The violations manager also checks that the heuristic repair algorithm can only update the scheduling plan if the update results in a lower constraint violations value for the scheduling plan or at least in a smaller makespan. The pseudocode implementation for this can be seen in codeblock 7. The displayed pseudocode gets executed every time the algorithm tries to find a new assignment for a conflicting item.

Algorithm 7 Constraint violation update mechanism used by the heuristic repair algorithm

```
1: for all possibleNewStartValues do
2:     createNewPossibleAssignment()
3:     violatorUpdate ← violationsManager.tryViolatorUpdate(...)
4:     addToPossibleUpdates(violatorUpdate)
5: end for
6: bestUpdate ← getBestPossibleUpdate()
7: violationsManager.updateViolator(bestUpdate)
```

Basically, the algorithm tries to find the best possible assignment from all possible start values *[l.1-5]* and then transmits the changes, that this new assignment brings, to the violations manager *[l.6-7]*. The best new assignment for a conflicting item is found by creating a new assignment for every reasonable start value *[l.2]* and then the violations manager computes how the constraint violations would change from that new assignment *[l.3]*. The resulting update-object is the delta between the current status of the violations manager and the status it would have after the new assignment. If the new assignment resulted in even more constraint violations, then the constraint manager would throw an exception and not create an update-object. If the violations manager receives an update-object it has created *[l.7]*, it will simply apply the changes that the object contains.

The update-object is an instance of the class `ViolatorUpdate` and contains the updated data for the violator in form of several instances of the class `PartnerUpdate`. They contain the new pre-computed values for the constraint violations of the rescheduled item.

The job of keeping track of the different possible new assignments and updates generated by the violations manager is done by the class `ConfigurationsManager`, which can be seen in figure 3.7. Every new assignment that creates a violator update object is used to create a new instance of the class `Configuration`. The configurations manager then organizes these configurations by comparing them to each other and to a reference configuration. The reference configuration is the original assignment that the algorithm tries to improve. If the reference configuration is also the best configuration possible, then the algorithm will be stuck in a local optimum and has to try and escape it.

Figure A.3 shows how the configurations manager compares different configuration. This is used when the manager tries to determine which configuration is the best one. As can be seen in figure 2.4, the priority of the different attributes of the configuration are very important as they lead to different results. The approach used in the implementation is to give the makespan of the

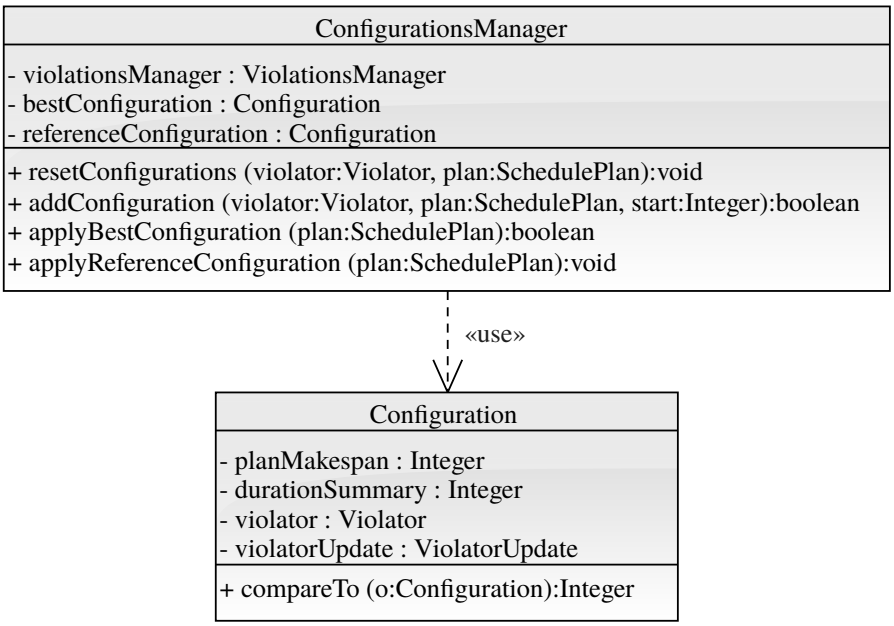


Figure 3.7: Configurations manager diagram.

scheduling plan the highest priority. This leads to better results, but also increases the amount of steps needed to find a feasible solution.

3.2.4 Dealing with local optima

Local optima are the biggest weakness of the algorithm, because it is very hard for it to escape from such a situation. Figure 3.8 shows an example for a constraint value landscape for different scheduling plans.

The higher a point is in the landscape, the less constraints are violated by the corresponding scheduling plan. Given that the algorithm is a hill-climbing approach, the arrows show the direction the algorithm will go to. For example, if the algorithm started at a point that is close to the marker C, then it would inevitably end at the marker C. The scheduling plan that corresponds to the point C is also a local optimum. But, since it is not a feasible solution, the algorithm can not stop there. Instead, it has to find a way to escape the local optimum.

There are several different approaches to solve the problem for general local search algorithms [8]. However, most of them do not meet the requirements stated earlier in this chapter. The following list shows some possible approaches and the reasons why they are not used in the implementation:

Randomized restart A very simple solution is to simply restart the algorithm with a different start configuration. Eventually, a start configuration will be found that does not result in a local optimum. For example, if the algorithm reaches the point A in figure 3.8 then it will

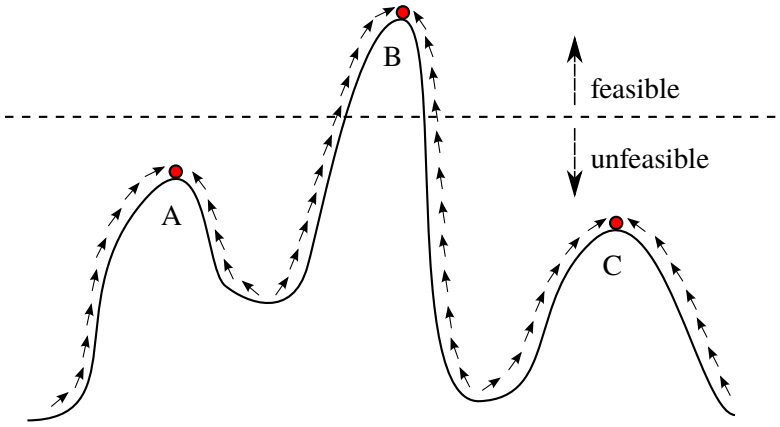


Figure 3.8: Constraint landscape.

simply restart the search at another random point of the landscape. Hopefully, with the new start point, the algorithm works its way to point B, which is a feasible solution.

However, this approach cannot determine if a solution exists at all. It might be the case that some of the constraints are faulty and contradict each other, in which case this approach will just restart the algorithm over and over again. Another reason not to use this approach is that it involves some kind of randomization. This would destroy the determinism of the original algorithm and make it harder to test the implementation.

Modify constraint weights The constraint violations each have different weights depending on the constraint and the violated item. The idea of this approach is to change the weight of the constraints for some schedules items. The constraints for such items can either be values less, in which case other items are more likely to be moved, or they are valued more, in which case they themselves are more likely to be moved.

No matter which method is chosen, the additional weights for the constraints change the form of the landscape. An example of this can be seen in figure 3.9. The blue line shows the landscape after applying the new constraint weights. The scheduling plan that corresponds to the point A in the old landscape corresponds to the point D in the new landscape. If the algorithm starts from this point D, it will be able to reach the solution at point B without becoming trapped at other local optima.

This approach, which is also called *breakout method* [34], has several drawbacks. For one, it is unable to solve many situations on its own [8, p.5]. The change of the landscape can create new local optima where the algorithm gets stuck again. In addition, implementing this requires many modifications on the violations manager and maybe even the constraints, which makes the implementation unnecessarily complex. Also, the weights, with which the constraints should be multiplied or divided, must be determined through testing and in the end it just adds more parameters to the algorithm.

Randomized bad moves Another way to escape a local optimum is to enable the algorithm to

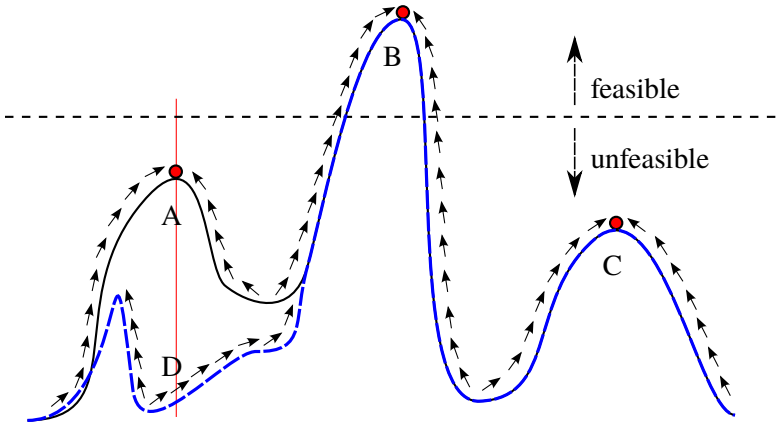


Figure 3.9: Constraint landscape with modified constraint values. The black line shows the original landscape, the blue one the landscape with revalued constraints.

make moves that result in a worse scheduling plan. A scheduling plan is considered worse if it has more constraint violations or a longer makespan.

For example, the algorithm could accept bad moves from time to time with a low probability. Basically, this is the same method the Simulated Annealing algorithm uses to escape local optima. However, this approach would add unwanted randomized behaviour to the implementation. Also, since the Simulated Annealing approach does not work well with scheduling problems such as these, it might be possible that this technique is not a good choice. Another major drawback is that the whole implementation focuses on the hill-climbing mechanism and it would add unnecessary complexity to enable the implementation to handle bad moves.

Domain value penalties Instead of changing the weight of the constraint violations, the assignments of a local optimum themselves can be penalized. That means that once a local optimum was reached, all the scheduled items get a penalty value added to them. This penalty is connected to their start value, so once they are reassigned they lose the penalty. This changes the shape of the constraint landscape similarly to the example in figure 3.9.

This method has the same drawback as modifying the constraint weights, except that is easier to implement. However, the main problem is that most of the time only very few scheduled items need to be reassigned to escape a local optimum. This approach, on the other hand, penalizes all items equally, whereas most of them do not need to be reassigned. In addition, the range of possible start values is rather big, so it might take a lot of reassignments and additional penalties until the local optimum is finally escaped.

None of the presented methods seem like good choices for the implementation, mostly because they were designed as general solution that work with all CSPs. This generic nature makes them very versatile, but it adds the drawbacks described before.

By making use of the problem structure, three solutions could be found that do not have the drawbacks of the generic approaches. More specifically, all of these three solutions still use the hill-climbing approach. However, all of them willingly trade an increased makespan for fewer constraint violations. Basically, they stretch the scheduling plan to escape the local optimum. The following list describes the different solutions in detail:

Move dependency tree Most of the time, local optima are created because the phase dependency constraints were violated. Therefore, a method that is guaranteed to solve phase dependency constraint violations is guaranteed to solve most local optima problems. This can actually be done quite easily and a pseudocode representation of the implementation is displayed in codeblock 8.

Algorithm 8 Escaping a local optimum by moving all dependent items

```
1: function MoveDependentItems
2:     root ← getMostViolatingAssignment()
3:     dependencyTree ← buildDependencyTree(root)
4:     unscheduleAllItemsFromTree(dependencyTree)
5:     while traverseBreadthFirst(dependencyTree) do
6:         findNewAssignmentForCurrentTreeNode()
7:     end while
8: end function
```

An example how this method works is also displayed in figure 3.10. The first thing the algorithm does is to determine the root of the dependency tree *[l.2]*. This is always the item with the highest constraint violation values. In the example of figure 3.10 this can be either item C or item B, but item C was chosen to be the root node.

The next step is to build the actual dependency tree from the item that represents the root node *[l.3]*. The actual procedure to build the tree is not described here, but the pseudocode for it can be found in codeblock 10 in appendix B. In the given example, the dependency tree has item C as root node and item D as dependent node.

After that, the algorithm removes all items contained in the dependency tree from the scheduling plan *[l.4]*. All of them are then rescheduled in the order of their dependency. This is achieved by traversing the dependency tree in a breath-first order *[l.5-7]*, such that all items of one dependency level have been scheduled before the items of the next level are scheduled. Every item that is rescheduled gets assigned to the place where it violates the least constraints possible *[l.6]*.

Not only does this method solve the problem presented in the example, but it does indeed solve all local optima caused solely from violated phase dependencies. This is because all of the involved items are removed from the schedule and are guaranteed to be added in a new order that fits all phase dependencies. Furthermore, this approach does not introduce any bad moves, because either the result contains strictly less constraint violations or it is not accepted. In addition to all of that, the algorithm can be implemented very efficiently,

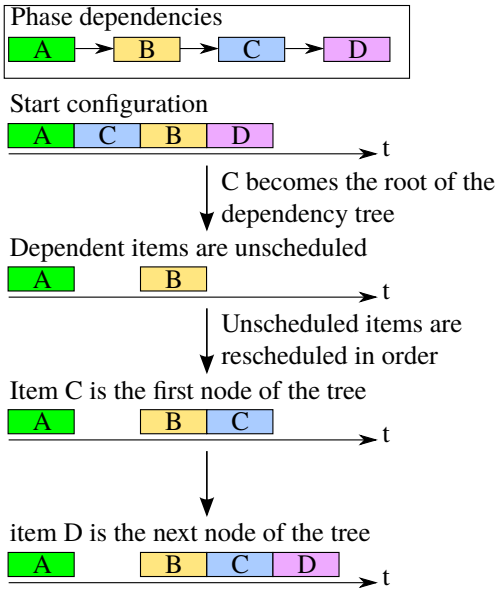


Figure 3.10: Example how to escape a local optimum by moving the dependency tree.

because it affects only the items connected to the root node of the dependency tree. This means that usually only very few items are moved and have their constraints recalculated.

Shift items to the right Although the dependency tree method works quite nicely, it still makes assumptions about the constraints used. However, this can be very problematic since the constraints can contain any programmable logic.

An example of this problem is depicted in figure 3.11. The example uses the following two constraints:

1. Item B must be executed immediately after item A. Note that this is not a phase dependency, but rather an arbitrary time-constraint. Therefore, it cannot be used by the dependency tree method.
2. Item B must not be executed on more than one lane at the same time.

Clearly, the second constraint is violated right from the start. However, there is no new assignment for the item B that does not violate the first constraint. The result is a local optimum that is not created by any phase dependency constraint and that can therefore not be solved by the dependency tree method.

The solution to this problem is to shift all violated items so far to the right that they are guaranteed not to be scheduled at the same time as any other items. The pseudocode for this approach is displayed in codeblock 9.

The algorithm works recursively and shifts items to the right until no additional constraint violations are created. The first thing it does is to actually shift all violated items *[l.2]*. The

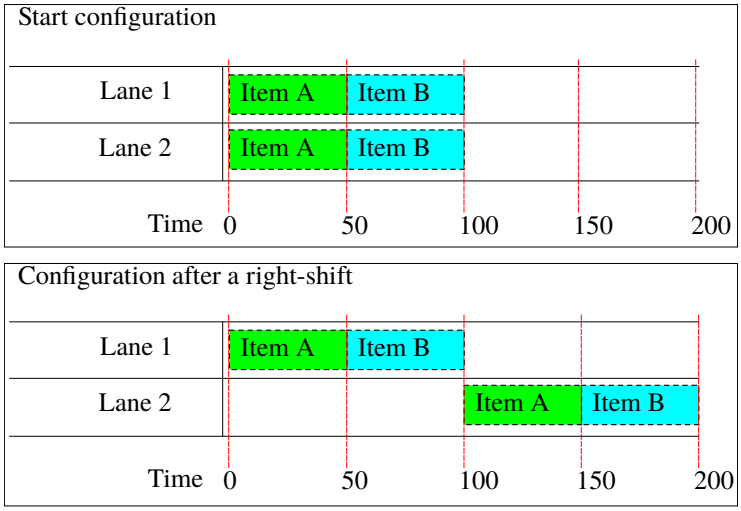


Figure 3.11: A local optimum that can not be solved with the dependency tree method. Given are the following two constraints: 1. Item B must be executed immediately after item A, 2. Item B must not be executed on more than one lane at the same time.

Algorithm 9 The recursive algorithm used to shift items to escape a local optimum

```
1: function RecursiveShiftAndLock(itemsToShift)
2:     shiftItemsRight(itemsToShift)
3:     lockShiftedItems()
4:     newViolations ← getNewViolatedItems()
5:     if containsLockedItem(newViolations) then
6:         abortBecauseOfCircularConstraint()
7:     end if
8:     recursiveShiftAndLock(newViolations)
9: end function
```

amount they are shifted to the right is the original makespan of the scheduling plan. In the example of figure 3.11, item B is originally scheduled to start at 50 before it gets shifted to the right by a value of 100, leading to a new start value of 150.

The next thing the algorithm does is to lock all shifted items in place *[l.3]*, because a circular constraint might otherwise lead to an endless loop. Once the items locked, the algorithm checks if the shifting of the items has created new constraint violations. If this is the case, the algorithm retrieves a list of all the items involved in these new constraint violation *[l.4]*. Of course, the items that were just shifted by the algorithm are not included in this list. In the example of figure 3.11, the shifting of item B creates a new constraint violation with item A, so item A is registered by the algorithm as the next item to shift.

The algorithm then checks if one of these new items to shift has already been shifted and locked before *[l.5-7]*. If this is the case then the constraints refer to each other in a circular way and it is impossible for the algorithm to find a solution.

If the items pass the check for circular constraints then the next step will be a recursive call where they are the next items to be shifted *[l.8]*. In the example of figure 3.11, item A is passed on and then also shifted to the right, leading to the final configuration displayed in the figure.

This method can be seen as a more general version of the dependency tree method that does not care about the nature of the constraints used, but also creates worse results. The problem with this method is that it must shift the items with a value equal to the makespan of the scheduling plan, leading to a doubling of the makespan in the worst case. Therefore, it is only used if the dependency tree method fails to escape the local optimum.

Shift items to the left Although the right-shift method works with more constraints than the dependency tree method, there are still many possible configurations that are local optima and cannot be solved with it. For example, figure 3.12 shows a start configuration that can not be solved by the previous methods, given the following constraint:

1. Item C must be executed before all items A_i from any lane i.

Clearly, item C violates the constraint two times in the start configuration, as it is scheduled after the items A_1 and A_2. The tree dependency method does not solve this problem, because there are no phase dependencies violated. Also, the right-shift method does not work, because moving item C to the right does not solve the existing constraint violations.

The solution is to do the opposite of the right-shift method and shift the violated items to the left. However, the smallest start value for a scheduled item is zero and negative values are not allowed. Therefore, *all* items in the scheduling plan are shifted to the right prior to the actual left-shift. The example in figure 3.12 shows this as the second step of the procedure.

The algorithm used to shift the items to the left is basically the same as the one used for the right-shift. The pseudocode is the same as the one displayed in codeblock 9, except for

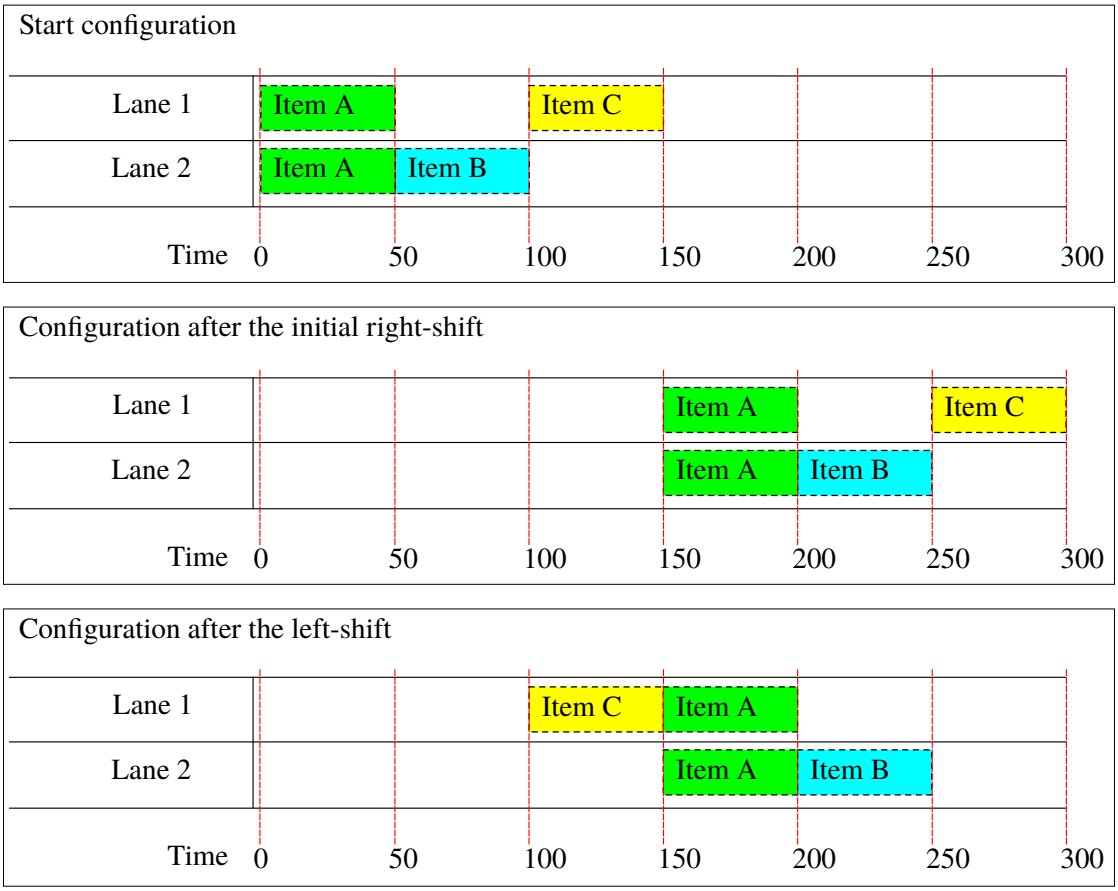


Figure 3.12: A local optimum that can not be solved with the right-shift method. Given is the following constraint: 1. Item C must be executed before all items A_i from any lane i.

one thing: instead of shifting the items to the right by the value of the makespan *[l.2]*, they are shifted to the left.

It is pretty obvious from the result displayed in figure 3.12 that this method increases the makespan by a large value. In addition, it creates a large gap at the beginning of the scheduling plan. The reason for this is that initially all items are shifted to the right, but only a small group of items is then shifted back to the left. Therefore, this method is only used if all other approaches failed.

Although these three approaches are enough to escape almost all local optima, they still expect a certain constraint behaviour. For example, they expect that binary constraints only consider the positions of the checked items relative to each other. If a binary constraint changed its behaviour depending an the absolute start value of the items then these methods would most certainly fail.

If such a situation occurs and a local optimum can not be escaped then the algorithm will terminate with an exception. This is the reason why the hardware commissioning software will still use the original prototype solution as a backup scheduler.

Chapter 4

Implementation and Testing

Although the implementation is pretty straightforward when given the design, it still has some noteworthy details. In addition, this chapter describes some of the optimizations and problems, as well as the tests used to ensure a high code quality.

The implementation itself is done in Java, because the whole LHC hardware commissioning software is written in Java and the integration should be as easy as possible. Java is an object-oriented programming language with a syntax similar to C++. Its applications are compiled into bytecode that can then be executed on a virtual machine (JVM) on any platform. This point, among others, helped Java to become one of the most popular programming languages in use.

4.1 Implementation details

This section highlights some of the implementation details as well as some additional features that are not part of the original design.

Immutable data objects All data model objects, like `ItemToSchedule` or `Violator`, are implemented to be *immutable*. This means that they cannot be changed once they are created. An example for a standard Java class that implements this behaviour is the class `String`. Although a string has many methods for manipulation, like splitting or concatenation, they all create a new object with the changed content and leave the original object alone.

Although it is a bit more effort to implement them, immutable object grant a number of different advantages:

1. They are easy to test and use. Testing is obviously simplified, because state-changes of these objects cannot occur and have therefore not to be tested. It also simplifies the use of these objects, because their instances can be shared freely without needing a special copy-constructor or clone-method. There is just no need for a defensive copy when sharing them. This also makes them great candidates to be cached, which can greatly reduce memory consumption.

2. They are thread-safe. This is a huge advantage if parts of the algorithm should run in parallel. The current implementation only has a small part of the implementation parallelized, but it benefits from it nevertheless.

3. They can be used as hash-table keys. If they were mutable, they would change their hash-code after each mutation, making them unstable keys for a hash-table.

So, making these objects immutable is a good choice, as it helps to improve the code quality, which is one of the main requirements of the implementation.

Viewer application To ease the development of the implementation and to visualize the results, a small viewer application was developed with the use of Java Swing. A screenshot of this viewer can be seen in figure A.4.

It represents the different scheduled items as half-transparent colored blocks. It also displays the different lanes and time-marks for every start or ending of a scheduled item. Since all of the displayed items are vector-based, it can zoom in and out to display even the largest scheduling plans in detail.

Another feature is its ability to visualize the different steps of the algorithm. This is a great way to quickly spot errors and eases the debugging work. The different steps are directly recorded by the algorithm by taking a snapshot after each iteration. Each snapshot is a simple copy of the scheduling plan at that point of time.

Although the viewer is a great help for the implementation of the algorithm, its use does not stop there. It would also be a nice addition to the existing hardware commissioning GUI, because at the moment there is no way to visualize the result produced by the scheduling algorithm. Since this GUI also uses Java Swing, it should be very easy to add the viewer as an additional feature.

Fixed items When scheduling magnet tests, the scheduling algorithm also has to be able to consider tests that have already been started. Of course, the algorithm must not be able to move them around in the scheduling plan to assign them a later execution time. If they were reassigned then it would be possible for the algorithm to bypass constraints for these tests. For example, the algorithm could decide to start another test on the same system and schedule the running test to start later.

To solve this problem, the class `SchedulePlan` allows to fixate certain scheduled items by using the `fixateItem`-method. Once an item has been fixed, it can never moved or unscheduled from the plan. Any attempts to do so will result in an exception.

The fixed items are treated as every other item when it comes to constraint violations. The repair algorithm must consider this and not try to move them around, even if they violate the most constraints.

Also, the methods to escape local optima might not able to work correctly with fixed items. The dependency tree method cannot unschedule and reschedule them, even if they are a node in the tree. The right-shift and left-shift methods also cannot move them around, which could cause them to find no solution.

The algorithm is not prepared to check if two fixed items violate some binary constraint with each other. In theory, this can never happen as only those tests are started that comply with the

constraints. If, however, such a situation happened then the algorithm would simply be unable to produce a solution.

Start configuration The start configuration has a great impact on the performance of the algorithm. A good initial scheduling plan can completely prevent local optima and reduce the runtime of the algorithm to a minimum. However, finding such a good initial plan is almost as challenging as finding an actual solution.

There are many possible approaches to create a start configuration. The following possibilities have been implemented, but showed a poor performance:

- Every item is scheduled to start at the time zero: this creates a lot of constraint violations right from the start, because all of the scheduled items are overlapping. It takes many unnecessary moves just to solve these initial violations.
- Every item is scheduled to start at a random time: depending on the items and the constraints used, this approach also creates many constraint violations right from the start. The problem is that the random positioning violates dependencies between different tests most of the time.

  In addition, it easily creates local optima that have a negative impact on the result of the algorithm.

The algorithm currently used to construct the initial plan is rather simple, but the best solution tested so far. All of the items are placed as close as possible to the start value of zero without overlapping any other items. Basically, each of the different lanes has a chain of items scheduled for it. Although this does not consider the constraints between items of different lanes, it satisfies the dependency constraints for the initial plan.

4.2 Run-time analysis and optimizations

A number of optimizations have occurred during the implementation process. The table 4.1 and the graph in figure A.5 shows the great impact that these modifications have on the runtime of the algorithm.

Table 4.1: Execution times [ms] after different improvements. Each lane contains one item.

No. Lanes	Version#1	Redesign	Violators	Constraint-Map	Predictions	ForkJoinPool
1	110	100	94	63	32	31
101	4490	2162	1157	876	1020	1094
201	29605	12504	4249	2691	2016	1627
301	106519	48787	10608	7119	5394	4643
401	256064	120299	25767	16317	11617	8331

The test case used to measure the execution times after every modification of the implementation is very simple: for every lane, a single item must be scheduled and no item must be executed

at the same time as any other test. Therefore, every item is linked with every other item and it enforces the algorithm to check every binary constraint for every item after every move. This extreme coupling of items through constraints is not very common in a real-world application, but it is very helpful to quickly spot performance issues and memory leaks.

The implementation requires $\mathcal{O}(n^2)$ memory due to the fact that data-structures like the constraint-map[1] are used. In the worst case, these data-structures connect every test with every other test, leading to a quadratic memory consumption.

The execution time depends largely on the input, the constraints used and the start configuration. Although the current execution time is significantly lower than that of the first version, especially for a larger input, the *lower bound* for the execution is still $\Omega(n^2)$. It is not possible to implement the algorithm with a smaller lower bound, because every scheduled item has to be checked with every other scheduled item at least once, otherwise it cannot be guaranteed that all the constraints are fulfilled.

It is not easily possible to give an *upper bound* for the execution time. Analysis of the test case used to create the table 4.1 suggests an execution time close to $\mathcal{O}(n^3)$. However, this test case is not very representative for the normal use of the algorithm and it is easy to create an example with a far worse execution time.

The implementation of the algorithm includes a number of optimizations, the most important of which are explained in the following paragraphs.

4.2.1 Cancel violator update

As described in chapter 3.2.3, new possible assignments for a scheduled item must be checked by the violations manager using the `tryViolatorUpdate()`-method. If a new assignment violates less constraints than the current one, then the violations manager will create an update-object containing the data about the new assignment. If, however, the new assignment violates the same or even more constraints, then the violations manager will throw an exception and stop to check any remaining constraints for the new assignment.

This is better than an approach where the violations manager has to check every new assignment completely, because it needs significantly less checks of binary constraints.

4.2.2 Constraint predictions

Following the previous optimization to stop checking binary constraints of a bad new assignment, it is even better if no constraints have to be checked at all. To achieve this, every binary constraint is able to provide a limited prediction that can then be used to dismiss some new assignments without actually checking the constraints for them. These predictions have to be generated just once at the start of the algorithm and can then be used to predict any possible constellation.

To do this, every binary constraint is able to return an instance of the `ConstraintPrediction`-class when given two items to schedule as parameters. The UML class diagram for the prediction can be seen in figure 4.1.

[1] See chapter 3.2.3 or figure 3.6

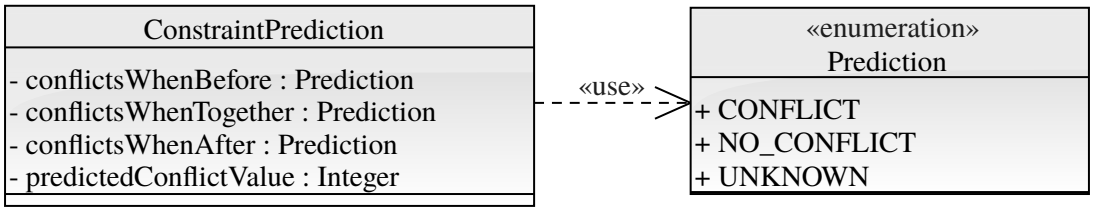


Figure 4.1: UML class diagram for the constraint prediction class.

Given two items, item A and item B, a constraint has to predict what will happens if if the items are arranged in different positions relative to each other. To do so, it is assumed that item A is fixed and item B can be moved around. There are three different cases that need to be predicted:

1. Item B is positioned *before* item A. This means that the end-time of item B must be smaller or equal to the start-time of item A.
2. Item B and item A have the same start-time, meaning that they are scheduled *together*.
3. Item B is positioned *after* item A. This means that the start-time of item B must be smaller or equal to the end-time of item A.

For each of these cases, the constraint has to predict one of the following outcomes:

1. There will certainly be a constraint violation. The constraint also has to predict how high the violation value will be.
2. There will certainly be no violation.
3. It is impossible to predict if there will be a violation.

So, each constraint prediction can be used to check the outcome of a constraint evaluation for two scheduled items. This means that every item could have its own collection of all the predictions from related items. This would be similar to the constraint map displayed in figure 3.6 or it would even be simple to just add it to the constraint map itself. However, there would be no advantage over checking n predictions instead of checking n constraints.

The solution to this problem is to not store all the predictions separately, but to aggregate them into a single prediction. To do this, each prediction is transformed into instances of the class `Block`. More specifically, the three different prediction cases mentioned above are transformed into one `BeforeBlock`, at least one `MiddleBlock` and one `AfterBlock`. The UML diagram for these classes can be seen in figure 4.2.

An example for this block construction from a constraint prediction is displayed in figure 4.3. Given are the following two items:

1. Item A with a duration of 100. This is the reference item whose start is considered as zero on the time-line.

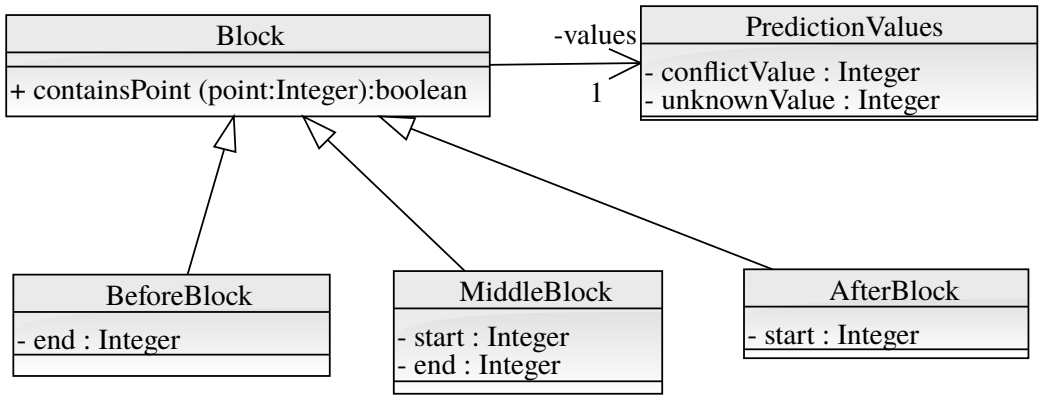


Figure 4.2: UML class diagram for the prediction block classes.

2. Item B with a duration of 200. This is the movable item which can be placed relative to item A.

For these two items, the constraint prediction is the following:

1. If item B is placed before item A then the constraint will be violated.
2. If item B starts together with item A then the constraint will not be violated.
3. If item B is placed after item A then the constraint will not be violated.

The prediction blocks created from this data can be seen as a time-line. The block α covers the time-line from $-\infty$ to the block's end point, the block β cover the time-line from the block's start point to ∞ and the block γ covers everything in between.[2] Therefore, each point on this time-line corresponds to exactly one block which contains the predicted values from the constraint. To find the predicted constraint value for a scheduled item B, all that has to be done is to search the block that corresponds to the relative start time of item B and then retrieve its values.

For example, if item B is scheduled to start 20 time-units after item A then the constraint will not be violated. This is because the point 20 on the time-line corresponds to the block γ which has a violation value of zero.

With this in mind it is easy to understand how the three blocks α, β and γ are constructed. The block α has to end at the point -200, because if item B starts anywhere in the range from $-\infty$ to -200 then the prediction states that the constraint will be violated. Similarly, the block γ has to start at the point zero, because if item B starts together with item A or after it, then the prediction states that the constraint will not be violated. The block β fills the gap in between, but it has an unknown value, because the prediction does does actually state what will happen when item A and item B are partially overlapping. For example, if item B is scheduled to start ten time-units before item A, then the constraint might be violated or not.

[2] Of course, the actual implementation can only schedule items up to a time that is equal to the biggest allowed integer value and not to infinity.

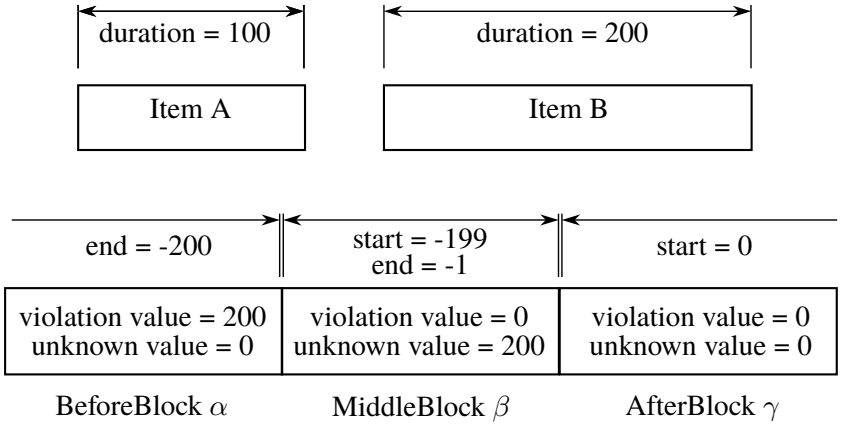


Figure 4.3: Block construction from a constraint prediction. Given are the item A with a duration of 100 and item B with a duration of 200. The constraint predicts a violation only if item B is placed before item A.

The creation and management of all the predictions and blocks is done by the class `Predictor` whose UML diagram is shown in figure 4.4 along with the classes it uses. The `Predictor` class is used by the violations manager to predict conflicts for a scheduled item by calling the `predictConflicts`-method.

To do this, the predictor uses a *prediction map* that maps every item to schedule to an instance of the `PredictionData` class. This prediction data keeps track of all the `PredictionBlocks` relevant for this item it is mapped to. Every instance of the `PredictionBlocks` class is basically one of the time-lines described in the example above.

The main purpose of this whole block-based approach is to be able to merge all the different predictions into a object that combines all of the information. This is exactly what the `aggregate`-method does. It takes a number of `PredictionBlocks` objects as input and merges all of them into one single object. The actual procedure to merge the blocks is not described here, but the pseudocode for it can be found in codeblock 11 in appendix B.

Figure 4.5 displays how two `PredictionBlocks` objects are aggregated into a new one. Every single block in the aggregated object is created by taking two blocks of the original two `PredictionBlocks` objects that have overlapping time values and then combine their constraint values. For example, the displayed block $\beta 3$ is created by combining the values of the blocks $\beta 1$ and $\alpha 2$, which overlap exactly in the time from -199 to -151.

It might be helpful to understand how all of the described parts work together by showing how they are used during the execution. The following scenario describes the steps taken when the algorithm tries to determine if a new assignment for a scheduled item is valid or not. The corresponding UML sequence diagram is shown in figure A.6.

1. The `ViolationsManager` calls the `predictConflicts`-method of the `Predictor` to determine if it has to check any constraints at all for the given scheduled item.

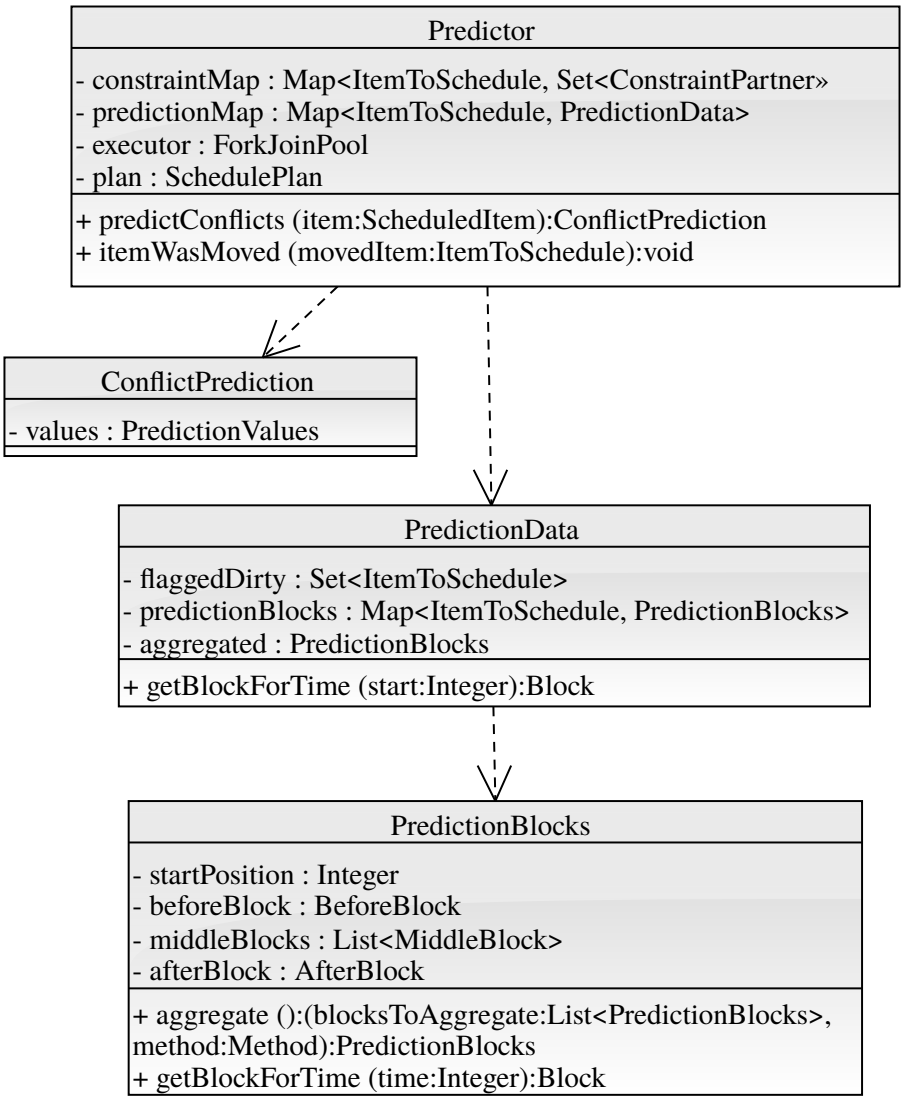


Figure 4.4: UML class diagram for the prediction management classes.

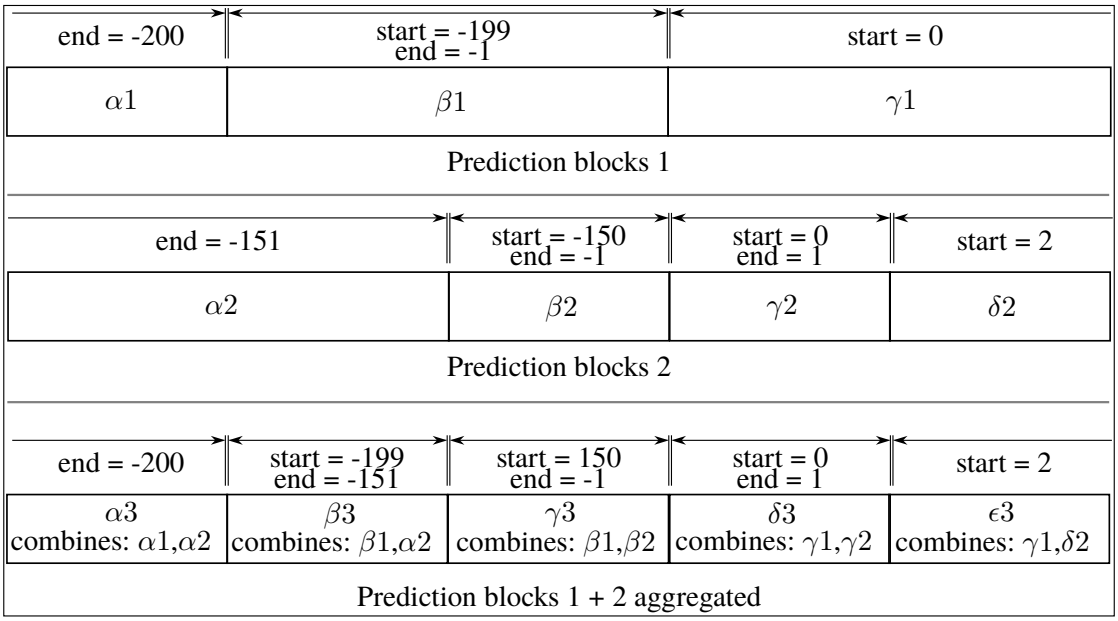


Figure 4.5: Two `PredictionBlocks` objects are aggregated into one.

2. The `Predictor` retrieves the correct `PredictionData` object for the given scheduled item. It then calls the `getBlockForTime`-method of this object with the start-time of the scheduled item as parameter.

3. The `PredictionData` creates an aggregation of all the stored `PredictionBlocks` objects, if not already done before. It then calls the `getBlockForTime`-method of the aggregated object to retrieve the `Block` object for the start-time of the scheduled item.

4. The values contained in the retrieved `Block` object are the predictions requested by the violations manager. They are packed into a wrapper object of the class `ConflictPrediction` and returned to the manager.

It is important to note that the prediction of constraints cannot be used to circumvent them, even if the prediction states that no constraint is violated. If the prediction does not automatically exclude a new assignment, then all the constraints are checked for it regardless of the prediction. This means that the prediction could be completely ignored by the algorithm and the result would be the same.

4.2.3 Parallelization

The constraint prediction mechanism is a great optimization of the implementation as it drastically lowers the amount of constraint checks needed, especially as the number of scheduled items increases. However, instead of checking all these constraints the implementation spends most of the time with the aggregation of the prediction blocks. This aggregation is a task that can very

easily be parallelized, but the subtasks of the aggregation are very small and the creation of a thread would take longer than actually running the subtask itself.

A new feature of Java 7 is the class `ForkJoinPool` that was created specifically for such purposes. Instead of just being a thread-pool it can very efficiently execute small tasks by also managing a task-pool. Each subtask is added to the task-pool that can be accessed by a thread pool. Each tread can quickly execute several tasks and grab new ones from the pool after completion.

By executing this rather small part of the implementation in parallel, the execution time could be noticeably lowered, as can be seen in table 4.1. This is especially true as the number of tests increases, because then the number of blocks to aggregate also increases and the overhead to manage the different threads becomes less important.

4.3 Testing

It is very important to test the implementation thoroughly, because every missed bug in the scheduling algorithm could lead to equipment damage. It is not only the algorithm itself that has to be tested, but also the adapter that connects the algorithm to the scheduling framework. This adapter has many different responsibilities:

- Create the data-objects for the scheduling algorithm from the test objects of the test framework.
- Use grouping constraints to find groups among the tests.
- Map the results of the scheduling algorithm back to the original tests.
- Monitor the operations of the scheduling algorithm to switch back to the backup strategy in case of problems.

Despite its great importance, the tests used to verify the adapter are not described in this thesis, as the main focus lies on the implemented algorithm itself.

All of the tests are created as unit tests with *JUnit*, which is a well-established test framework for Java. In addition, the code analysis tool *Atlassian Clover* is used to determine the code coverage of the unit tests. All of the critical parts of the implementation must have a test coverage[3] of 100%. However, even a high test coverage does not guarantee that there are no bugs in the implementation. The advantage of the coverage tool is that it helps to identify parts of the implementation that were not tested at all.

The tests used for different parts of the implementation differ from each other depending on the complexity of the implementation. The main data classes, like `ItemToSchedule`, are each tested with simple unit tests only. This is sufficient enough, because these classes are just data containers and mainly consists of *getters* and *setters*.

[3] *Clover* evaluates the code using the *branch coverage* approach, also known as C1 coverage. This means that every conditional statement that creates a branch must evaluate to $true$ and $false$ at least once.

Unit tests Usually, unit tests are used to check a single *functional unit* of an application. This might be a single method or a complete module, but usually a unit test is used to check the functionality of a single class. This is done by comparing the result for a certain input with a defined output. Normally, there is some kind of specification that states how the result should look like and that is tested in this way.

It is virtually impossible to proof that the tested program is correct using unit tests, because this would require one test for every possible input. Even for small and simple programs the number of possible inputs is just too big. When the program environment is also part of test, for example when checking system portability or compatibility to other programs, then it becomes truly impossible to check every possibility with unit tests.

Therefore, unit test cases are selected to cover the most common use-cases as well as the ones that are the most likely to cause errors. For the scheduling framework, the common use-case includes scheduling a few tests using some simple constraints as well as scheduling many tests with more complex constraints. The inputs that are likely to cause errors include empty inputs or unusual constraints that create local optima. It is also important to check that faulty inputs, like constraints that create circular dependencies, do not break the implementation.

Although the scheduling framework could not be tested in production, there are also unit tests that use the actual production data. This includes test data taken from the hardware commissioning database as well as all of the constraints used during the commissioning.

The problem with these unit tests in general is that most of the time it is very hard to define what the result should be. It is easy to check if all of the constraints are satisfied, but it is very hard to make assumptions how the tests are to be aligned or how big the makespan must be.

It is not the aim of the algorithm to compute the optimal solution, but rather an approximation. However, for most of the bigger unit tests the optimal solution is not known and there is no bound how much the approximation may divert from it. Therefore, for these unit tests the algorithm was executed prior to writing the unit test and the result was used as a reference for the makespan. So, checking the makespan during a unit test is not used to check the specification, because there is no specification on the makespan. Nevertheless, checking the makespan is useful, because it shows if a change of the implementation has a negative impact on the created scheduling plans.

Static code analysis Apart from the unit tests, several static code analysis tools are used to ensure a good code-quality. The tools used are *Checkstyle*, *PMD* and *FindBugs*. Although these analysis tools are not able to verify that the code works as intended, they still have many advantages.

They enforce that the code complies with certain style rules to be easily readable. Furthermore, they are able to point out parts of the code that might cause bugs, like exposing internal data structures of an object or trying to dereference a null-pointer. This semantic analysis of the code is very powerful and is able to find many bugs that would otherwise have gone unnoticed.

The disadvantage of these tools is that they are very sensitive and create many false negatives. But even though it sometimes takes a lot of time to use them, they definitely help to improve the quality of the code.

Chapter 5

Conclusions

It is not possible to use the implementation of the algorithm in a real hardware commissioning before this thesis is finished. The LHC will be still be operational for several months and it is not possible to use the LHC hardware for testing during normal operation. Therefore, the evaluation of the implementation is only based on artificial tests.

The aim of this thesis was to create and describe a scheduling algorithm that is able to handle to high workload during the LHC hardware commissioning. Several different possibilities were discussed for the algorithm and in the end the Heuristic Repair approach was chosen to be implemented. Its implementation was described and how the problems of the algorithm, like being trapped in a local optimum, were solved. Also, some of the tests used to verify the implementation were described.

The implemented algorithm was successfully integrated into the test framework used for the LHC hardware commissioning. At the same time, the implementation is generic enough to be used in other projects as well. It is able to schedule any items with arbitrary, custom constraints very efficiently.

During the test runs the implementation has proven to be able to schedule a high amount of items. It takes about ten seconds to schedule 1000 tests for the LHC. Although this result is good enough to to deploy the implementation with the test framework, there are still many possible optimizations that can lower the execution time.

A special viewer was implemented to display the result of the scheduling algorithm. Even though it was originally just a debugging tool, it was implemented in the GUI of the test framework to display the schedule during the hardware commissioning.

5.1 Outlook

There are still many possible improvements to be made to the implementation. For example, the algorithm could group the different items into clusters depending on their constraint relations. It could then schedule each cluster separately and combine the results into one scheduling plan. This task could easily be parallelized to further improve performance.

It would be really helpful if the implementation was able to reuse previous results. Starting with a scheduling plan that is almost feasible is a lot faster than starting from scratch. The start

configuration is one of the most important factors for the runtime of the Heuristic Repair algorithm. During a hardware commissioning campaign, most of the time there will be only small changes, like adding or removing a test from the scheduling plan. These changes could simply be applied to previous results before starting the scheduling algorithm.

The viewer provided to display the results could be further improved by actually allowing to manually change the scheduling plan. It could also somehow display which constraints act on which items, because for the users it is not always clear why some tests are not allowed to start.

The source-code of the implemented algorithm will be released as open-source, so that other projects are able to reuse it. As a side-effect, this might also help to further improve the code quality and add new features.

Appendix A

Figures

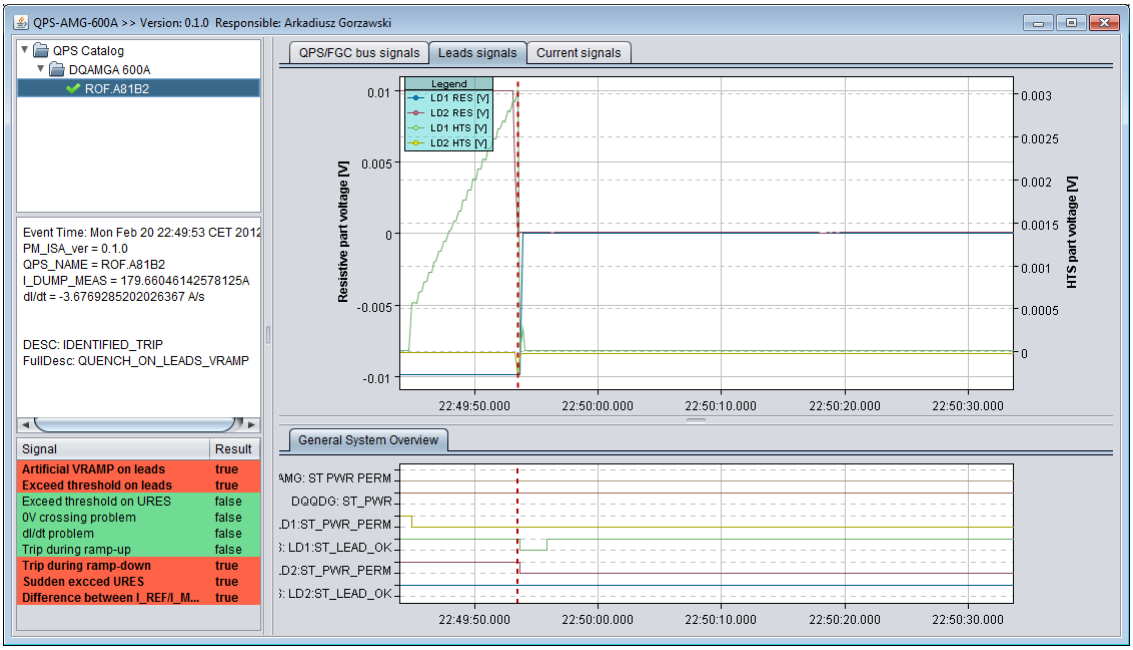


Figure A.1: Electrical signal measurements taken during a magnet test

Figure A.2: Test system user interface showing the current test status of several systems

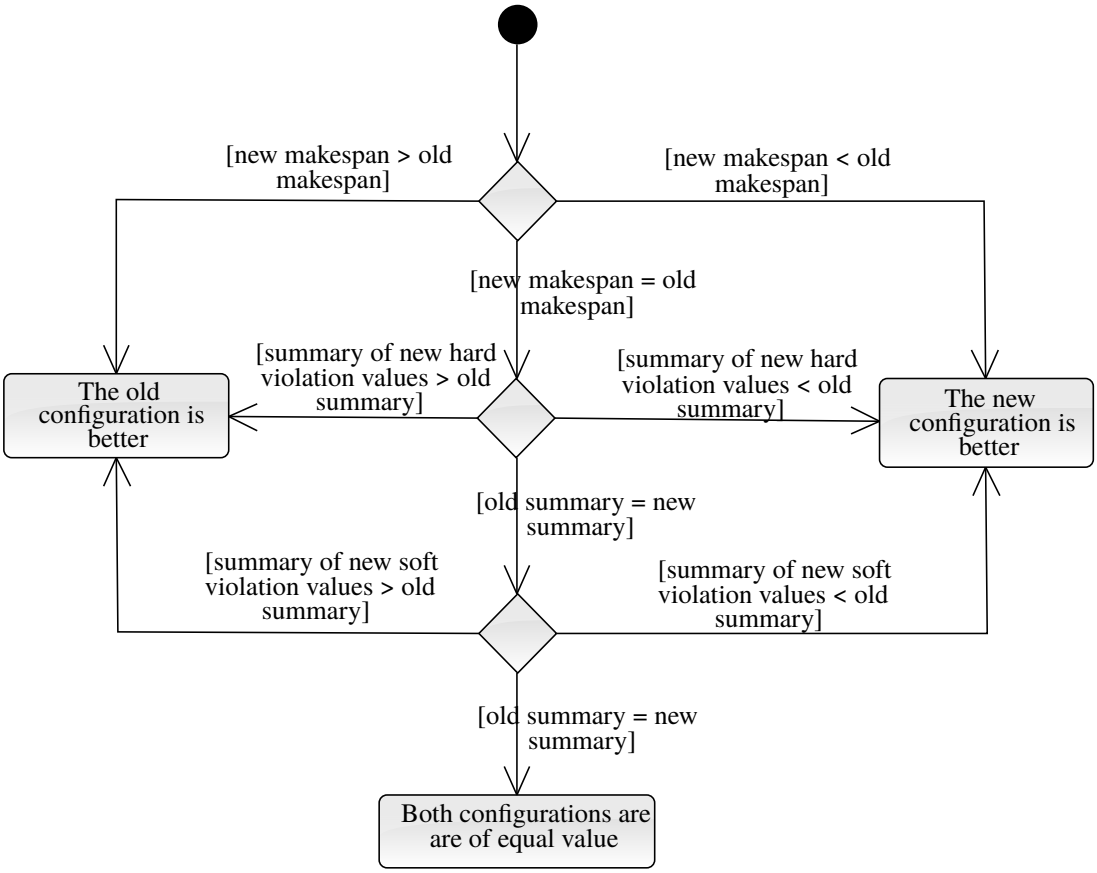


Figure A.3: Configuration comparison diagram.

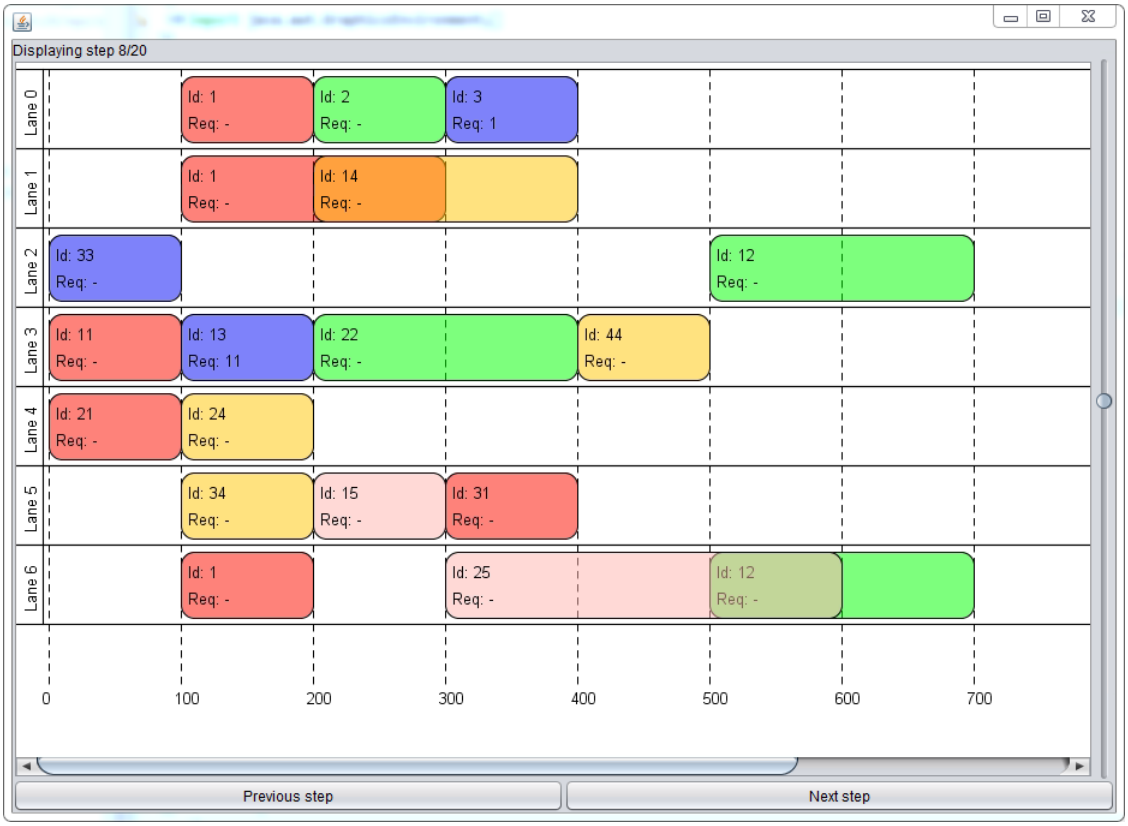


Figure A.4: Viewer used to visualize the steps of the Heuristic Repair algorithm

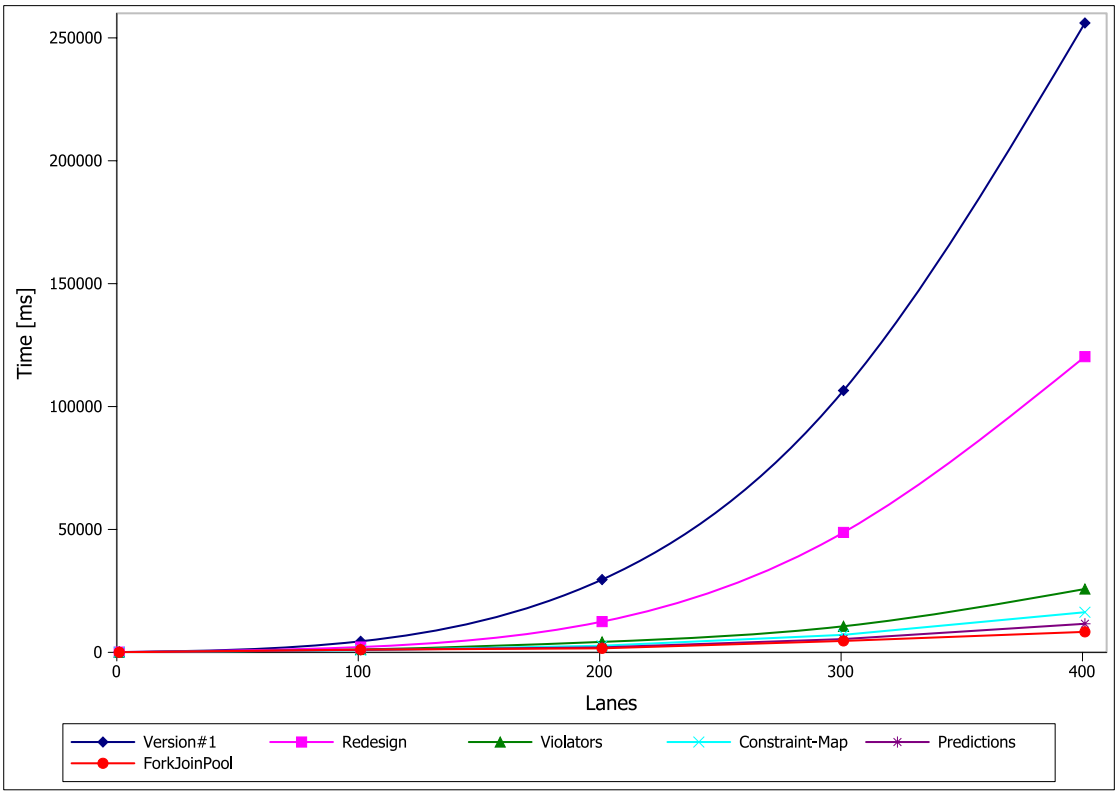


Figure A.5: Execution time of the implementation after different improvements.

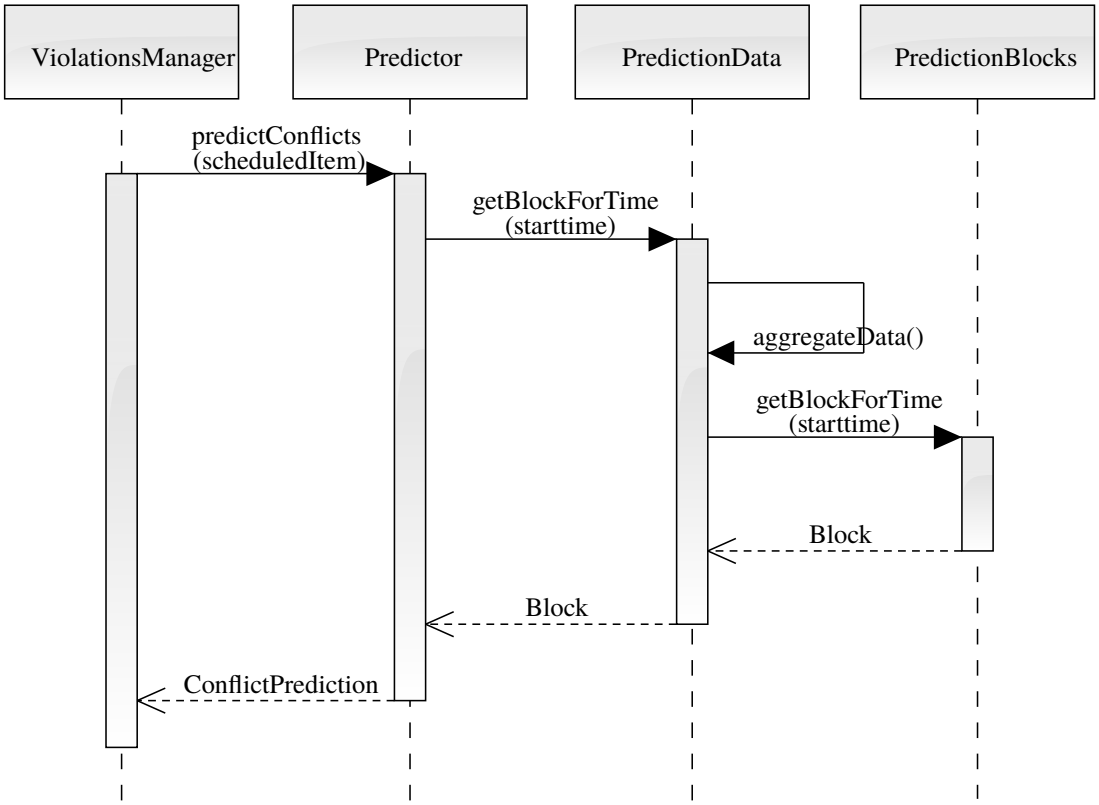


Figure A.6: Sequence diagram of a constraint prediction for a new scheduled item.

Appendix B

Sources

Algorithm 10 Building the dependency tree for a scheduled item

```
1:  function BuildDependencyTree(currentItem, tree, currentLevel)
2:      if not tree.containsNode(currentItem) then
3:          tree.addNodeAtLevel(currentItem, currentLevel)
4:      else if currentLevel > tree.getLevelForNode(currentItem) then
5:          tree.updateLevel(currentNode, currentLevel)
6:      end if
7:      for dependentItem in getDependentItems(currentItem) do
8:          buildDependencyTree(dependentItem, tree, currentLevel + 1)
9:      end for
10: end function
```

Algorithm 11 Aggregating a list of prediction blocks

```
1:  function Aggregate(blocksToAggregate)
2:      gatherAllStartAndEndTimes(blocksToAggregate)
3:      endTime ← getNextEndTime()
4:      createNewBeforeBlock(endTime, blocksToAggregate)
5:      while hasNoNewAfterBlock() do
6:          startTime ← getNextStartTime()
7:          if hasNoMoreStartTimes() then
8:              createNewAfterBlock(startTime, blocksToAggregate)
9:          else
10:             endTime ← getNextEndTime()
11:             createNewMiddleBlock(startTime, endTime, blocksToAggregate)
12:         end if
13:     end while
14: end function
```

Algorithm 12 The extended version of the prototype scheduling solution

```
1:  function SCHEDULEPOSSIBLETESTS(allTests)
2:      possibleTests ← removeImpossibleTests(allTests)
3:
4:      for all system : allSystems do
5:          scheduledTests{system} ← pickPossibleTest(system, possibleTests)
6:      end for
7:
8:      while changed(scheduledTests) do
9:          for all scheduledTest : scheduledTests do
10:             if contradictsGroupConstraint(scheduledTest) then
11:                 remove(possibleTests, scheduledTest)
12:                 replacement ← pickPossibleTest(scheduledTest.system, possibleTests)
13:                 if replacement ≠ null then
14:                     scheduledTests{scheduledTest.system} ← replacement
15:                 else
16:                     remove(scheduledTests, scheduledTest)
17:                 end if
18:             end if
19:         end for
20:     end while
21:
22:     return scheduledTests
23: end function
24:
25: function PICKPOSSIBLETEST(system, tests)
26:     for all checkedTest : tests_system do
27:         isValid ← true
28:         for all constraint : binaryConstraints do
29:             if contradictsBinaryConstraint(checkedTest, constraint) then
30:                 isValid ← false
31:             end if
32:         end for
33:         if isValid = true then
34:             return checkedTest
35:         end if
36:     end for
37:
38:     return null
39: end function
```

Bibliography

[1] E.H.L. Aarts and J.K. Lenstra. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003. ISBN: 9780691115221.

[2] M. Affenzeller. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*. Numerical Insights. CRC Press, 2009. ISBN: 9781584886297.

[3] M.J. Atallah and M. Blanton. *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*. Applied Algorithms and Data Structures Bd. 1. Chapman & Hall, 2009. ISBN: 9781584888222.

[4] W. Atmar. "Notes on the simulation of evolution". In: *Neural Networks, IEEE Transactions on* 5.1 (1994), pp. 130 –147. ISSN: 1045-9227.

[5] A. Auger and B. Doerr. *Theory of Randomized Search Heuristics: Foundations and Recent Developments*. Series on Theoretical Computer Science. Singapore: World Scientific, 2011. ISBN: 9789814282666.

[6] V. Baggiolini et al. *A Sequencer for the LHC ERA*. Tech. rep. CERN-ATS-2009-114. Geneva: CERN, 2009. URL: https://cdsweb.cern.ch/record/1215886.

[7] P. Banerjee, M.H. Jones, and J.S. Sargent. "Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors". In: *IEEE Transactions on Parallel and Distributed Systems* 1 (1990), pp. 91–106. ISSN: 1045-9219.

[8] Muhammed Basharu, Inés Arana, and Hatem Ahriz. "Escaping Local Optima: Constraint Weights vs. Value Penalties". In: *Research and Development in Intelligent Systems XXIV*. Ed. by Max Bramer, Frans Coenen, and Miltos Petridis. Springer London, 2008, pp. 51–64. ISBN: 978-1-84800-094-0.

[9] Eric Biefeld and Lynne Cooper. "Bottleneck identification using process chronologies". In: *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 1*. IJCAI'91. Sydney, New South Wales, Australia: Morgan Kaufmann Publishers Inc., 1991, pp. 218–224. ISBN: 1-55860-160-0.

[10] Jacek Blazewicz, Wolfgang Domschke, and Erwin Pesch. "The job shop scheduling problem: Conventional and new solution techniques". In: *European Journal of Operational Research* 93.1 (1996), pp. 1–33.

[11] José M. Cecilia et al. "Enhancing data parallelism for Ant Colony Optimization on GPUs". In: *Journal of Parallel and Distributed Computing* (2012). ISSN: 0743-7315.

[12] CERN. *How the LHC works*. 2008. URL: http://public.web.cern.ch/public/en/lhc/HowLHC-en.html.

[13] B. Coppin. *Artificial Intelligence Illuminated*. Jones and Bartlett Illuminated Series. Jones and Bartlett Publishers, 2004. ISBN: 9780763732301.

[14] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. "The Ant System: Optimization by a colony of cooperating agents". In: *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics* 26.1 (1996), pp. 29–41.

[15] Marco Dorigo and Thomas Stützle. "The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances". In: *Handbook of Metaheuristics*. Ed. by Fred Glover and Gary Kochenberger. Vol. 57. International Series in Operations Research and Management Science. Springer New York, 2003, pp. 250–285. ISBN: 978-0-306-48056-0.

[16] Dr.R.Umarani and V.Selvi. "Article: Comparative Analysis of Ant Colony and Particle Swarm Optimization Techniques". In: *International Journal of Computer Applications* 5.4 (2010). Published By Foundation of Computer Science, pp. 1–6.

[17] S. Edelkamp and S. Schroedl. *Heuristic Search: Theory and Applications*. Morgan Kaufmann. Elsevier Science, 2011. ISBN: 9780123725127.

[18] D.E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Artificial Intelligence. Addison-Wesley Pub. Co., 1989. ISBN: 9780201157673.

[19] R. L. Graham. "Bounds for certain multiprocessing anomalies". In: *Bell System Technical Journal* 45 (1966), pp. 1563–1581.

[20] J.W. Greene and K.J. Supowit. "Simulated Annealing Without Rejected Moves". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 5.1 (1986), pp. 221–228.

[21] Darrall Henderson, Sheldon Jacobson, and Alan Johnson. "The Theory and Practice of Simulated Annealing". In: *Handbook of Metaheuristics*. Ed. by Fred Glover and Gary Kochenberger. Vol. 57. International Series in Operations Research and Management Science. Springer New York, 2003, pp. 287–319. ISBN: 978-0-306-48056-0.

[22] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. Bradford Books. MIT Press, 1992. ISBN: 9780262581110.

[23] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007. ISBN: 9780321455369.

[24] Mark Jones and Prithviraj Banerjee. "An improved simulated annealing algorithm for standard cell placement". In: *Proceedings of the International Conference on Computer Design* (1988), pp. 83–86.

[25] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680.

[26] M. Kolonko. "Some new results on simulated annealing applied to the job shop scheduling problem". In: *European Journal of Operational Research* 113.1 (1999), pp. 123 –136. ISSN: 0377-2217.

[27] Vipin Kumar. “Algorithms for constraint-satisfaction problems: a survey”. In: *AI Mag.* 13.1 (Apr. 1992), pp. 32–44.

[28] Yu-Kwong Kwok and Ishfaq Ahmad. “Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm”. In: *Journal of Parallel and Distributed Computing* 47.1 (1997), pp. 58 –77. ISSN: 0743-7315.

[29] C. Lecoutre. *Constraint networks: techniques and algorithms*. John Wiley & Sons, 2009. ISBN: 9781848211063.

[30] Amnon Meisels. “Distributed Search by Constrained Agents”. In: *Intelligent Distributed Computing V*. Ed. by F. Brazier et al. Vol. 382. Studies in Computational Intelligence. Springer Berlin / Heidelberg, 2012, pp. 5–9. ISBN: 978-3-642-24012-6.

[31] Daniel Merkle, Martin Middendorf, and Hartmut Schmeck. “Ant Colony Optimization for Resource-Constrained Project Scheduling”. In: *IEEE Transactions on Evolutionary Computation*. Morgan Kaufmann, 2000, pp. 893–900.

[32] Steven Minton et al. “Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems”. In: *Artif. Intell.* 58.1-3 (1992), pp. 161–205. ISSN: 0004-3702.

[33] Steven Minton et al. “Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method”. In: *Proceedings of the eighth National conference on Artificial intelligence - Volume 1*. AAAI’90. Boston, Massachusetts: AAAI Press, 1990, pp. 17–24. ISBN: 0-262-51057-X.

[34] Paul Morris. “The breakout method for escaping from local minima”. In: *Proceedings of the eleventh national conference on Artificial intelligence*. AAAI’93. Washington, D.C.: AAAI Press, 1993, pp. 40–45. ISBN: 0-262-51071-5.

[35] D.L. Poole and A.K. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press, 2010. ISBN: 9780521519007.

[36] Siamak Sarmady. *An Investigation on Genetic Algorithm Parameters*. Universiti Sains Malaysia, 2007. URL: http://sarmady.com/siamak/papers/genetic-algorithm.pdf.

[37] Bart Selman, Hector Levesque, and David Mitchell. “A new method for solving hard satisfiability problems”. In: *Proceedings of the tenth national conference on Artificial intelligence*. AAAI’92. San Jose, California: AAAI Press, 1992, pp. 440–446. ISBN: 0-262-51063-4.

[38] B. Smith. “Using heuristics for constraint satisfaction problems in scheduling”. In: *Advanced Software Technologies for Scheduling, IEE Colloquium on*. 1993, pp. 2/1 –2/3.

[39] Rok Sosic and Jun Gu. “A polynomial time algorithm for the N-Queens problem”. In: *SIGART Bull.* 1.3 (Oct. 1990), pp. 7–11. ISSN: 0163-5719.

[40] Tony White, Simon Kaegi, and Terri Oda. “Revisiting Elitism in Ant Colony Optimization”. In: *Genetic and Evolutionary Computation — GECCO 2003*. Ed. by Erick Cantú-Paz et al. Vol. 2723. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, pp. 199–199. ISBN: 978-3-540-40602-0.

[41] David H. Wolpert and William G. Macready. “No Free Lunch Theorems for Optimization”. In: *IEEE trans. on Evolutionary Computation* 1.1 (1997), pp. 67–82. URL: http://axon.cs.byu.edu/~martinez/classes/678/Papers/Wolpert_NLFoptimization.pdf.