# RESOLVER

BOOLEAN SATISFIABILITY SOLVER BASED ON A GENETIC ALGORITHM

## Methodologies

**Team Imperium**

Dewald de Jager
Ernst Eksteen
Vignesh Iyer
Regan Koopmans
Craig van Heerden

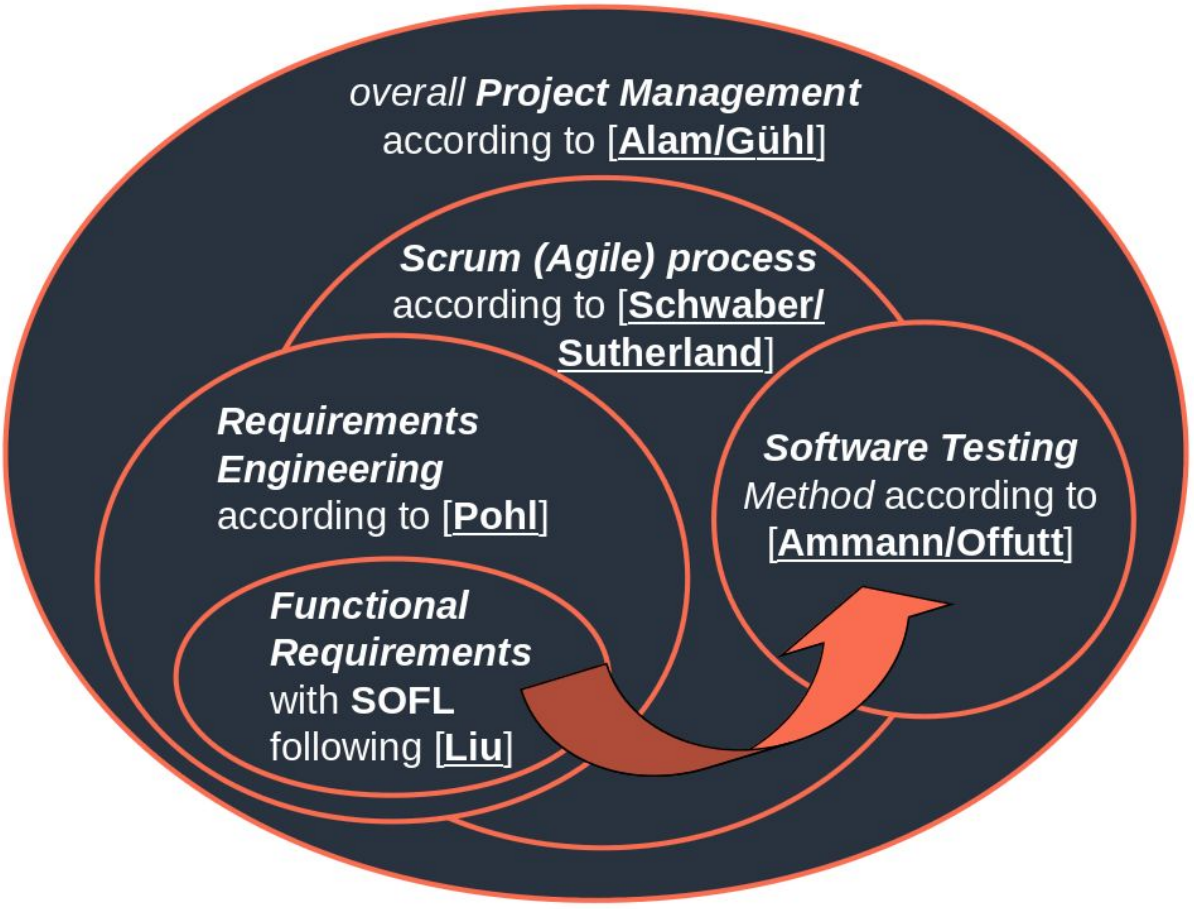
October 19, 2017

# Table of Contents

# Overview

Software development requires not only a process but also a methodology or development method. A methodology defines the steps of how to carry out the activities of a software process. It is an implementation of the process. This document outlines the software methodologies that we followed in each of the software life cycle activities. It ascertains to the fact that our project conforms to various academic and industry standards.

The various methodologies and how they relate can be seen in the Venn diagram below.



# Project Management Methodology

## The Scrum Agile Process

Agile is a development methodology that accommodates recurrent alterations to scope by practising iterative development against well-defined objectives, over fixed length time-frames called “sprints”.

Our choice in this project management methodology over others has allowed our project to adapt favourably to change, as our understanding of the problem domain grew and the requirements of our clients became more clear.

Our team has made scheduled releases of the software as it has progressed through development. These scheduled releases have given our project a sense of steady advancement and urgency that may not have been there otherwise. These releases also act as snapshots of the project, which can be returned to so as to review change over time. Although our project management has prefered short-term measurable goals, major releases allow us to gauge the larger more intangible properties of the software as a product.

We have had regular correspondence with the clients both in person and through electronic means. A mailing list was established early in the life of the project, which has aided communication greatly. We had several meetings with the SSFM research group, which gave us opportunities to ask questions and demonstrate progress.

Following the Agile paradigm, our documentation has been iterative. All members of the team have contributed to a collective understanding of the requirements of the software requested by the clients. These assumptions have been returned to, questioned, and amended several times. We have designed the system symbolically, and have used use-case diagrams to define the behaviour of the system and its components in an abstract manner.

We maintained a fixed-length sprint of 2 weeks, which were bounded by structured sprint meetings. These meetings identified goals that were met during that previous sprint, defined goals for the next sprint, and reflected on the current general health of the project. Minutes were taken of each meeting and published to a central repository (GitHub Wiki). This action gave us ownership of the goals we set. This management structure was often supplemented by half-sprint meetings, when such meetings were deemed necessary, often occurring near the time of an important evaluation event.

As a team we have been deliberate in focussing our efforts to those particular features that create the most value to our client. Portioning our work this way has kept the feedback loop of development short, and allowed us to reaffirm our progress at regular intervals.

The project has had a number of critical deployment targets. Firstly, there is the software application itself, which is packaged for three major operating systems. Additionally we have two SAT solving instances that run on distinct remote servers. These servers support our functionality of our team's website (which is served and maintained on one of the two servers), and demonstrate a crucial property of the software, namely that an interface can connect to a geographically separated SAT instance over a network.

## Project Management Tools

To supplement our project management methodology, we made use of Waffle - an automated project management tool. This tool enabled integration with issues and pull requests that we made on GitHub, allowing us to track our workflow very effectively. It provided us with the ability to capture the status of our project visually. The use of this virtual scrum board our philosophy of "making the process open source to complement code that is already open source".

# Requirements Engineering Methodology

It is very important to follow a tried-and-tested methodology as a guide for what should be documented and how, and to avoid making common mistakes. In this specification we use the requirements engineering methodology as laid out in *Requirements Engineering: Fundamentals, Principles, and Techniques*. This methodology complements our Agile software engineering methodology as it promotes continuous requirements engineering. There are many benefits to treating requirements engineering tasks as cross-lifecycle activities, but the most beneficial to this project are that the requirements stay up to date and are used systematically. The responsibilities of the system and its components are made very clear.

The Pohl requirements engineering methodology identifies three type of requirement artifacts:

1. Goal-Oriented Requirements
2. Scenario-Oriented Requirements
3. Solution-Oriented Requirements

Goal-oriented requirements are specified at a very high level and state what the system is supposed to achieve. Scenario-oriented requirements describe concrete examples of satisfying or failing to satisfy goals; They are meant to illustrate the goals with more detail and are typically defined through use of user stories and use-case diagrams. Finally, solution-oriented requirements define the data, functions, behaviour, quality and constraints of the system. These requirements have been divided further into three solution-oriented requirement artifacts:

- Functional Requirements: Define the services that should be provided by the system

- Data Dictionary: Defines the data types of data flowing through the system
- Behavioural Requirements: Defines algorithmic information that cannot be captured by functional requirements, such as execution-order

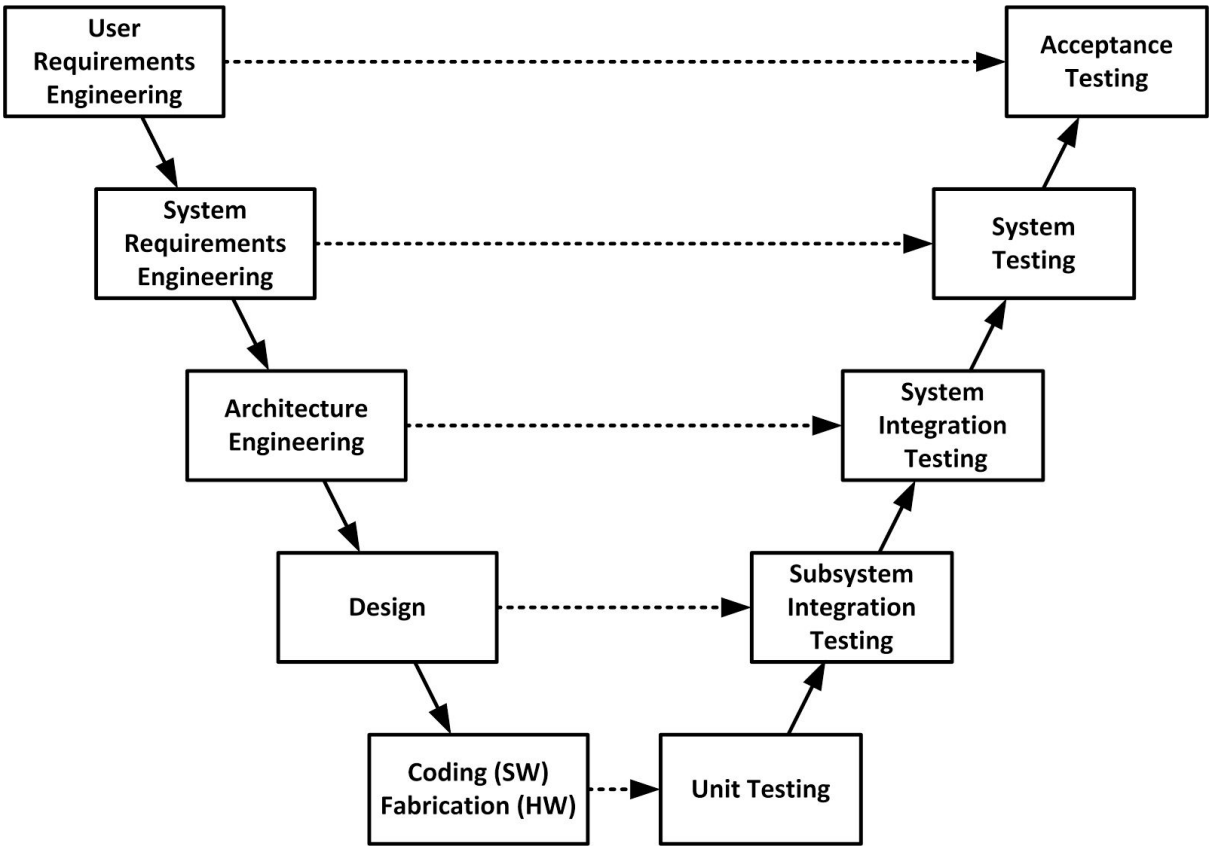
Due to the applications that Resolver will be used for and their scientific nature, our client requested that we provide a semi-formal proof of the system. This proof has been done using a well-known formal language used for software verification called the Structured Object-Oriented Formal Language (SOFL). SOFL defines its own formal syntax and also a graphical language for defining a system. The focus of our proof is on the various processes and data in the core algorithm of the system. The inputs, outputs, preconditions, postconditions, invariants and relationships between the components are specified in SOFL. The proof (and a description of the language) as well as the other artifacts are explained in more detail in the Resolver Software Requirements Specification.

# Software Testing Methodology

Software testing is a practical engineering activity that is essential to producing high-quality software. We use the software testing methodology presented in Introduction to Software Testing by Ammann and Offutt for designing test inputs, producing test case values, running test scripts and analyzing and reporting the results. The methodology places a large emphasis on treating testing activities as a lifecycle-wide activity and encourages designing tests concurrently with each of the development activities regardless of whether the software artifact is in an executable state. This allows us to identify defects in our design decisions early on in the project reducing the overall time and cost.

## Test Level Descriptions

Our testing process is divided into different testing levels, where each testing level corresponds to a software development activity. This is widely known in literature as “The V Model”. This model illustrates how the testing levels relate to each of the lifecycle activities.




An overview of each of the testing levels is given below:

## Unit Testing

Unit testing verifies that the smallest entities (units) can function correctly when isolated from the rest of the units. A unit is the smallest entity which can independently exist and is typically one or more contiguous program statements with a name that other parts of the system use to call it. In the case of Resolver, units are implemented as functions and methods. The unit test plans are developed primarily during the detailed design phase. These plans are executed to eliminate bugs at code level or unit level. This level is the lowest level of testing and is done in a very isolated manner. We make use of techniques such as input space partitioning to ensure that generalisations of the different types of cases are found, including corner cases. We also use automated testing tools to assist in running the unit tests as part of regression testing.

## Integration Testing

Integration Test verify that units created and tested independently can coexist and that the interfaces between modules have consistent assumptions and communicate correctly. Since Resolver has relatively few modules, we have decided to merge the integration and module testing phases. Module testing assesses each module in isolation and verifies that the units that each module is composed of interact correctly and use the shared data structures correctly. Both of these testing activities are typically the sole responsibility of the system development team.

### System Testing

System testing verifies that functional and quality requirements have been met and that the system as a whole meets its specifications. System tests plans are primarily developed during the architectural design phase. It is assumed that the different parts of the system work individually and the whole application is tested for its functionality, interdependency and communication. Load and performance testing as well as stress testing are done at this stage. This testing phase is usually done by a team other than the developers but due to the nature of this project we have assigned the roles among ourselves and dedicated time for this purpose.

### User Acceptance Testing

The goal of user acceptance testing is to ensure that the software does what the user wants. User acceptance test plans are therefore primarily developed during the Requirements Engineering phase and is the testing must involve users, the product owner and/or individuals with strong domain knowledge. User acceptance testing is performed in a user environment that resembles the production environment, using realistic data. Our data for this testing purposes is acquired from the data sets of global SAT competitions and encodings from the System Specifications and Formal Methods research group at the University of Pretoria.

## Test Field Descriptions

Each test case in our software testing report document is described by these fields:

1. **Test Case ID**: A unique identifier for the test case
2. **Purpose**: A short textual description of the purpose of the test case
3. **Expectation**: A short textual description of the anticipated observation for the specified input
4. **Pass Criterion**: The condition that needs to be satisfied in order for the test case to be considered as “passed”
5. **Input**: The values that were used as input for this test case
6. **Observation**: A short textual description of what was observed when the unit being tested was executed - This should be completely objective and no judgement should be made according to the pass criterion
7. **Judgement**: Whether the test case was passed or not (Yes / No)

## Testing Tools

Testing is a very important part of the software development lifecycle and as such requires a lot of time and effort. To assist in maintaining high quality standards and implementing good software testing practices, we make use of a few vital tools.

The majority of the project is developed in Python and as such the unit tests have been implemented in Python. We use a unit testing framework called ***unittest*** that is provided by the Python standard library to run unit tests. This framework allows us to automate tests,

share setup and shutdown code among tests, aggregate related tests into collections and maintain independence between the code of the tests and the reporting framework.

In addition to ensuring all our code abides by our coding standards (The PEP8 standard), the JetBrains' PyCharm integrated development environment provides tools for running code coverage tests and reporting statistics. We have set goals of achieving 100% node coverage (Equivalent to 100% statement coverage) and reaching 90% edge coverage. Path coverage is practically impossible because of the large number of loops in Resolver resulting in a countless number of paths. For this reason, loops are manually tested using the methods suggested by Beizer in *Software Testing Techniques*.

PyCharm also includes a profiler that gives us insightful statistics on run-time performance which aided us in identifying bottlenecks, making optimisations and gathering statistics for stress tests. The profiler also populates a graphical call graph allowing us to see how many function calls were made, how long they took and the relationships between the functions.

The Travis continuous integration tool allows us to merge small code changes into the codebase frequently rather than merging in a single, large change at the end of the lifecycle. We hope to achieve higher-quality code by developing and testing in smaller increments. Travis allows real-time monitoring of tests as they execute, is highly configurable, can run tests concurrently and integrates nicely with our communication platforms such as Slack so that we receive updates in real-time.

# References


Alam, Daud, and Uwe F. Gühl. *Project-management in Practice*. Springer-Verlag Berlin An, 2016.

Ammann, Paul, and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

Beizer, Boris. *Software testing techniques*. Dreamtech Press, 2003.

Liu, Shaoying. *Formal Engineering for Industrial Software Development: Using the SOFL Method*. Springer Science & Business Media, 2013.

Pohl, Klaus. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

Sutherland, Jeff, and Ken Schwaber. "The scrum guide." *The Definitive Guide to Scrum: The Rules of the Game. Scrum. org* (2013).

van Rossum, Guido, Barry Warsaw, and Nick Coghlan. "PEP 8: style guide for Python code." *Python. org* (2001).