POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

ABCC - Automated Backward Compatibility Checker

Master's degree in
Computer Science And Engineering - Ingegneria Informatica

Author: **Joseph Di Salvo**

Student ID: 993406
Advisor: Prof. Alessandro Margara
Co-advisors:
Academic Year: 2022-23

Abstract

This thesis is about the development of a testing tool that ensures seamless backward compatibility during the release of two different company software. Tests are a crucial element used to verify behaviour of a software after some changes in the code, but unfortunately there are a lot of possible tests that can be conducted and understanding which best fits our goals is one of the first issues. There are tests to ensure performance during heavy workload, security through vulnerability scanning, single-units' correctness and many more. But we need a tool to efficiently test the main functionalities of the two company software, possibly with ad hoc solutions for what our purposes are.

Automated testing tools are designed to automate various aspects of the software development lifecycle. These tools are created to streamline, enhance, and expedite different stages of software development, maintenance, and deployment. Many tools exist and they are divided in groups depending on what the purpose of their test is.

That's why we developed a system composed by both our software and open-source public projects to have a unique tool able to test big industrial software, without external costs. We used the third-party integration approach to design back end that was able to communicate with both the company software and with both the provided front end. The result is the project ABCC, a tool to efficiently assess backward compatibility throughout new releases of a software. Our tool's testing feature falls into the branch of functional testing, overseeing all the aspects of the functionalities that we wanted to test on the property software.

Unlike existing solutions available online, our creation not only focuses on targeted compatibility testing but also leverages an integrated approach with minimal external dependencies. This amalgamation not only guarantees cost-effectiveness but also streamlines the intricate process of ensuring the enduring harmony of software evolution.

Abstract in lingua italiana

Questa tesi tratta dello sviluppo di uno strumento di testing che garantisce una compatibilità retroattiva senza soluzione di continuità durante il rilascio di due diverse applicazioni software aziendali. I test sono un elemento cruciale utilizzato per verificare il comportamento di un software dopo alcune modifiche nel codice, ma sfortunatamente esistono molti possibili test che possono essere condotti e comprendere quale si adatti meglio ai nostri obiettivi è uno dei primi problemi. Ci sono test per garantire le prestazioni durante carichi di lavoro pesanti, per la sicurezza attraverso la scansione delle vulnerabilità, per la correttezza delle singole unità e molti altri. Abbiamo però bisogno di uno strumento per testare efficacemente le principali funzionalità delle due applicazioni software aziendali, possibilmente con soluzioni ad hoc per i nostri scopi.

Gli strumenti di testing automatizzati sono progettati per automatizzare vari aspetti del ciclo di sviluppo del software. Questi strumenti sono creati per semplificare, migliorare e accelerare diverse fasi dello sviluppo del software, della manutenzione e della distribuzione. Esistono molti strumenti e sono suddivisi in gruppi in base allo scopo del loro test.

Ecco perché abbiamo sviluppato un sistema composto sia dal nostro software che da progetti pubblici open source per avere uno strumento unico in grado di testare software industriale di grandi dimensioni, senza costi esterni. Abbiamo utilizzato l'approccio di integrazione di terze parti per progettare il backend in grado di comunicare sia con il software aziendale che con il frontend fornito.

Il risultato è il progetto ABCC, uno strumento per valutare efficacemente la compatibilità retroattiva durante i nuovi rilasci di un software. La funzione di testing del nostro strumento rientra nella categoria del testing funzionale, che supervisiona tutti gli aspetti delle funzionalità che volevamo testare nel software proprietario.

A differenza delle soluzioni esistenti disponibili online, la nostra creazione non si concentra solo sul testing mirato della compatibilità, ma sfrutta anche un approccio integrato con poche dipendenze esterne. Questa fusione garantisce non solo la convenienza economica ma semplifica anche il complesso processo di garantire la duratura armonia dell'evoluzione del software.

Acknowledgements

First of all, I would like to thank prof. Alessandro Margara for having accepted to be my advisor and for his great support for this thesis redaction. The project was designed and implemented during my internship and my work experience in E80Group, so I would like to thank the company itself for giving me this opportunity.
Thanks to Roberto Olmi, manager of the ABCC project, for helping me in designing the system and especially for his continuous support during my experience. Thanks to Francesco De Mola, the leader of the Traffic Management Team, for his encouragement and for giving me the chance to work with the whole team. Thanks to Fabio for his support on the SmartCad software, helping me explaining and answering doubts through the development of the tool. Thanks to Christian for his support on the Smart software and for modifying some API endpoints for my purpose. Thanks to Francesco and Alessandro, my other two colleagues, who have directly and indirectly participated to the implementation of this project. It has been a pleasure working with all of you.
I would like to thank all my family and my friends, especially my mother, my father and my sister. You've always supported and helped me through these years.

Contents

List of Figures

1 | Introduction

The Automated Guided Vehicle (AGV) industry has experienced remarkable growth, transforming material handling processes across various sectors with its autonomous systems. As AGVs play an increasingly critical role in optimizing operations and supply chains, ensuring their flawless performance and safety becomes paramount. Similarly, in the fast-paced world of software development, seamless updates and releases are essential, but guaranteeing backward compatibility remains a significant challenge. Backward compatibility ensures that the newly introduced changes do not disrupt the functionality of previously supported versions, allowing users to seamlessly transition without encountering unexpected issues or incompatibilities.

The significance of backward compatibility cannot be overstated, as it directly impacts user satisfaction, customer loyalty, and the overall reputation of software products and companies. Therefore, in the realm of software engineering, the need for an efficient and reliable method to assess backward compatibility becomes increasingly vital.

This thesis endeavors to address the challenge of guaranteeing backward compatibility in software releases through the development of an innovative and ad-hoc automated testing tool. The primary objective is to create a comprehensive solution that can efficiently evaluate the compatibility of new software versions with their predecessors.

The proposed automated testing tool leverages a combination of static and dynamic analysis techniques to thoroughly examine the changes made during software development. By automatically generating test cases, the tool aims to identify potential points of conflict or incompatibility that could arise when users transition to the latest version.

The project is part of a collaboration with "E80 Group S.p.A.", an IT company with headquarters in Viano (RE). This company, among the other business, is specialized in the development of automated and integrated intralogistics solutions for manufacturers of consumer goods operating in the beverage, food, tissue and other sectors. The main systems produced by E80 Group include palletizing robots, a wide range of laser guided vehicles, high speed robotic stretch wrappers, pallet control systems, robotic labelers, layer picking and repacking solutions, and automated high-density warehouses. The project's

idea was about assessing backward compatibility, focusing on the Traffic Management branch of the company.

1.1. The Problem Context

The idea was to offer the company a basic testing tool for the two software that are in charge of managing the traffic of the AGV fleet across a whole factory. The problem was that transitioning to the latest version of a program required a human examination that was definitively time consuming and that, due to the millions of data to be examined, could have led to an oversight. Meaning that a flawed software could be released on the market, leading to AGV operations that could be pointless or wrong. That's why they were looking for an ad-hoc automated solution, with the help of some open-source technology too, to reduce the human error to the minimum and to know where their software behaved in a wrong or uncanny way. In the company I was inserted in the R&D Traffic Management team. This project was and is still in collaboration with the whole of the team as they are and will be the end users of this program. Initially, given some directives, I started writing a first draft of the code. As the tool began to take shape I started gathering feedback from the other members of the team, as each of them covers different topics, to adapt and improve the tool in order to include every essential aspect of the traffic management. Here the project ABCC (Automated Backward Compatibility Checker) started. It is a system that provides a CLI to launch the test and a GUI to both launch the test and change the program's configurations. Results given by the execution of both interfaces will be stored and made available for the end user.

1.2. RoadMap

This thesis is structured as follows: it will focus on the structure of the testing tool and on the requirements it has to match, as identifying the specific goals and objectives and defining the criteria for backward compatibility assessment. Nevertheless it is also going to make a brief survey of the available software, dealing both commercial software and open-source project with their pros and cons. In the next chapter there will be a list of all the project requirements. In chapter 3, a quick glimpse at the technologies available today and that we relied upon is accessible. While in chapter 4 there is an illustration of what are the recent technologies available that however we decided not to use. Chapter 5 will give a brief look at the related work by examining prior research and business strategies. Then, in chapter 6 there is a description of what are the design choices that we decided to adopt and the overall system architecture. In the 7th chapter details of the actual implementation, mainly about the critical parts of the system, the list of the main endpoints and the final user interface will be available for reading. Moving on, in chapter 8, there is an evaluation, based on the given requirements, of the final result. Finally, in the last chapter concluding considerations about the project and the future work will be made.

2 | Requirements

These requirements serve as a foundation for the development of an effective and robust automated testing tool that can systematically assess backward compatibility in software releases, providing developers and organizations with the confidence to deliver seamless updates to their users.

2.1. Test Case Generation

The automated testing tool should generate a diverse set of test cases that cover various usage scenarios of the software. Test cases should be designed to exercise critical functionalities, edge cases, and interactions with different components.

2.2. Immediate Response

The tool's goal is to return a response as fast as it can, whether it returns a positive result at the end of the test or it returns a negative result. In case a negative result is returned an error code will be shown, indicating what is and when the error occurred.

2.3. Representation of Data

Using just one data format, given the need to represent two different software each with different outcomes and outputs, would be impossible. More data formats would be necessary to represent and comprehend the challenges of this project:

1. JSON: Used for data and configuration storage

2. Xml: For a quick representation of the outcome of the program

3. db: Used as both input and output for the company software

4. txt: Generated by the company software

5. bin: Generated by the company software

2.4. SmartCad Tester

The tool should be capable of verifying that different test cases match the ones generated by SmartCad. This control is executed on a various type of files as there are multiple file generated with different extensions. This evaluation should be performed in a static analysis as method of testing.

2.5. Smart Tester

The tool should be capable of simulating real-world interactions and scenarios by executing the generated test cases on the new version of the software. Dynamic analysis should monitor the software's behavior during testing to detect unexpected behavior, errors, or deviations from the previous version.

2.6. Backward Compatibility Rules

The tool should support the definition of backward compatibility rules based on specific project requirements. These rules will serve as guidelines to determine the level of compatibility expected between the new and previous versions.

2.7. Reporting And Analysis

The automated testing tool should generate comprehensive and easily understandable reports highlighting compatibility risks and potential issues.

2.8. Scalability And Performance

The tool should be scalable to handle large and complex software projects efficiently. It should be optimized for performance to deliver timely compatibility assessments, especially for time-sensitive releases.

2.9. User-Friendly Interface

The tool should have a user-friendly interface that allows developers and quality assurance teams to:

1. Easily configure tests

2. View reports

3. Interpret results

It should require minimal training and provide clear documentation for ease of adoption.

2.10. Command-Line Usage

The tool should be able to be started through a Command-Line Interface. The commands must be short and clear, while the parameters should be limited to one or two but no more.

3 | Technologies

3.1. C#

C# is a modern, object-oriented, and type-safe programming language. C# enables developers to build many types of secure and robust applications that run in .NET. C# has its roots in the C family of languages.

C# is an object-oriented, component-oriented programming language. C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components. Since its origin, C# has added features to support new workloads and emerging software design practices.

3.1.1. Exception Handling

Exception handling is a crucial aspect of writing robust and reliable code. It involves the practice of identifying and managing runtime errors or exceptional situations that might occur during the execution of a program. These errors, often called exceptions, can arise due to various reasons such as invalid input, file not found, network issues, or division by zero. Properly implementing exception handling can prevent crashes, enhance user experience, and facilitate graceful recovery from unexpected scenarios. However it should not be used as a substitute for validating inputs or handling predictable errors. In our source code we heavily had to rely on it. Even if we tried our best to handle predictable errors some where beyond our capabilities: loading external DLLs as the one from SmartCad and receiving possible faulted JSONs through the Smart API leads to unpredictable errors. Another relevant aspect of Exceptions is that they are easily extensible, enabling us to create our own exception with each of them handing back different info that could help in understanding what went wrong.

3.1.2. LINQ and Functional Programming

LINQ (Language Integrated Query) and functional programming are two powerful concepts in modern programming, particularly in languages like C#. While they are distinct in nature, they often intersect and complement each other in practical applications. LINQ is a feature in C# (and other .NET languages) that provides a consistent query syntax for querying data from various sources, such as collections, databases, XML, and more. Its queries focus on what data to retrieve rather than how to retrieve it, promoting a more expressive and readable code and at the same time, being strongly typed, it catches errors at compile-time rather than runtime.
Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Another relevant feature of functional programming is that it encourages composing small, focused functions to create more complex behaviour, promoting code reuse and readability.
LINQ, while not purely functional, draws inspiration from functional programming concepts. LINQ promotes declarative code style, which aligns with the functional idea of focusing on what to achieve rather than how to achieve it. LINQ queries are also often composed of multiple smaller transformations, similar to functional composition. Additionally, LINQ methods like `Select`, `Where`, and `Aggregate` are based on functional programming principles. These methods operate on collections in a manner reminiscent of functional transformations, enabling concise and expressive code.
In our code we took advantage of LINQ as we had to deal with many collections, databases and XML documents. Along with Lambda functions we managed to fill the gap between traditional imperative programming and functional programming concepts. Coding this way allowed us to save time and to make the code more readable for future changes.

3.1.3. Asynchronous Operations

Asynchronous operations in C# provide a way to perform tasks concurrently without blocking the main thread of execution. This is particularly important for tasks that might take time to complete, such as I/O-bound operations (e.g., reading/writing files, making network requests) or CPU-bound operations (e.g., complex calculations). Asynchronous operations in C# enhance application performance and responsiveness by enabling tasks to execute concurrently without waiting for slow operations to complete, thereby optimizing resource utilization. In C#, asynchronous programming is commonly achieved using the `async` and `await` keywords.

3.1.4. HTTP Request and Response

Performing HTTP requests and handling responses in C# is commonly done using the `HttpClient` class. This class provides an easy and efficient way to make HTTP requests and receive responses. C# provides other libraries and approaches for making HTTP requests, but the HttpClient class is one of the most commonly used and versatile options. HttpClient can be used to interchange information with an API. Interacting with APIs using HttpClient involves sending HTTP requests, handling responses, deserializing JSON data, and potentially including authentication or custom headers. To understand the endpoints, request formats, and authentication mechanisms specific to the API you're working with it is useful to read the API Documentation.
As our interaction with the Smart software occurs through an API we relied on the HttpClient which is the most diffused and one of the most versatile.

3.2. SQLite

SQLite is an in-process library that implements a self-contained, server-less, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is the most widely deployed database in the world with more applications than we can count, including several high-profile projects.

SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. The database file format is cross-platform - you can freely copy a database between 32-bit and 64-bit systems or between big-endian and little-endian architectures. These features make SQLite a popular choice as an Application File Format.

SQLite is a compact library. With all features enabled, the library size can be less than 750KiB, depending on the target platform and compiler optimization settings. There is a trade off between memory usage and speed. SQLite generally runs faster the more memory you give it. Nevertheless, performance is usually quite good even in low-memory environments. Depending on how it is used, SQLite can be faster than direct file system I/O.

Most of the SQLite source code is devoted purely to testing and verification. An automated test suite runs millions and millions of test cases involving hundreds of millions

of individual SQL statements and achieves 100% branch test coverage. SQLite responds gracefully to memory allocation failures and disk I/O errors. Transactions are ACID even if interrupted by system crashes or power failures.

SQLite, while versatile and widely used, has its constraints. It faces challenges in scenarios requiring high concurrency due to its file-based locking mechanism. Write-intensive applications may experience slower performance because of its single-write thread design. Furthermore, SQLite isn't suitable for large-scale applications and lacks network access support, making it less viable for distributed databases.

As our applicative reads and writes just once per file, the concurrency obstacle won't affect us. Network won't either be a problem as the database files we need will be in-memory and will be accessed by one user at a time. Whilst we will write a lot of records to our database, they will be wrote using the most time efficient queries, avoiding multiple insert queries and using just one insert per table, significantly speeding up execution time.

Given these points, how they can be resolved and the fact this software was already being used by other programs in the office we opted for this solution.

3.3. NLog

NLog is a popular, flexible, and extensible logging framework for .NET platforms. It provides a robust and efficient way to log messages from applications, making it easier to monitor and troubleshoot software during development, testing, and production. NLog supports various logging targets, including files, databases, email, and external services, allowing developers to customize how and where log data is stored.

Key Features of NLog:

1. Configuration Flexibility: NLog offers a highly flexible and XML-based configuration system, enabling developers to define logging rules and targets easily. The configuration allows for dynamic changes at runtime without requiring application restarts.

2. Logging Targets: NLog supports multiple logging targets, such as files, console, event log, databases (including popular database systems like SQL Server, MySQL, and PostgreSQL), email, and more. This versatility allows developers to choose the most suitable target for their application's specific logging needs.

3. Rich Logging Levels: NLog supports various logging levels, including Trace, Debug, Info, Warn, Error, and Fatal, providing granularity in logging based on the severity of the message.

4. Contextual Logging: NLog allows developers to include contextual information in log messages, such as thread ID, timestamp, call site, and custom properties, to enhance the log's readability and usefulness.

5. Log Message Formatting: NLog supports customisable log message formatting with various placeholders, enabling developers to control the output format for each log entry.

6. Asynchronous Logging: NLog supports asynchronous logging, which improves application performance by offloading the actual writing of log entries to a separate thread or task.

7. Custom Log Layout Renderers: Developers can create custom log layout renderers to extend NLog's functionality, allowing the inclusion of application-specific information in log messages.

8. High Performance: NLog is designed for optimal performance and efficiency, minimizing the impact on the application's runtime.

9. Pluggable Architecture: NLog follows a pluggable architecture, allowing users to extend and customize its behavior by implementing custom logging rules, targets, and layout renderers.

10. NuGet [[4]] Package: NLog is available as a NuGet package, making it easy to add logging functionality to .NET projects and keep it up-to-date with the latest releases.

Overall, NLog is a versatile and reliable logging framework that provides valuable insights into application behavior and helps developers quickly identify and resolve issues. Its widespread adoption in the .NET community, active development, and extensive documentation make it a top choice for logging needs in various .NET applications.

3.4. SmartCad

SmartCad is the first company software that we come across, and is used to create and visualize the environment of the factory plant that we are observing. It heavily relies on Computer-Aided Design (CAD) technology to create intricate and detailed layouts of the building and its structures . These layouts are then seamlessly integrated into a database, specifically stored in SQLite (.db3) files, to ensure efficient organization and retrieval of design data.

SmartCad serves as a comprehensive tool for engineers and designers to conceptualize, visualize, and communicate their architectural ideas effectively. The software facilitates the creation of complex architectural layouts, all the autonomous guided vehicles and

last but not least all the paths, comprehensive of nodes and segments, that all the AGVs may follow to reach their destination. Each of these segments has its own rules for travel direction, e.g. if it must be tread facing forward or backward or both, and they may be overlaid in multiple points. It is also used to generate autoblocks, which consist in a safety shape surrounding the real shape of the vehicle. They are used to avoid two or more vehicles from crashing against each other. Another relevant feature is the creation of the files that will be exported to the actual AGV, written in machine code to guarantee that the vehicle can be able to read it. This last feature will save one or more files, depending on the AGV, in plain text or binary files. Many other functionalities are available but they are not relevant for our purpose, so we will just move forward.

Key Features and Workflow of SmartCad:

Design precision is a cornerstone of SmartCad. It allows for accurate measurements and annotations. This feature-rich application seamlessly integrates these layouts into a streamlined database system, specifically stored in SQLite (.db3) files.

The database integration serves as an efficient repository, systematically organizing each layout for easy retrieval, modification, and management. The architecture of the system ensures that designers can access their stored designs, review, edit, and continue work on ongoing projects without hassle.

In summary, the features we are planning to test in relation to SmartCad include the load and save processes, the export of machine code files for the AGVs, and the computation of autoblocks. As SmartCad is not provided of an API we will dynamically import the required DLLs and call the required methods be reflection.

3.5. Smart

Smart is a software application designed to accurately manage Automated Guided Vehicles (AGVs) traffic within manufacturing plants, there's also the option to run a plant in simulation mode to test the effectiveness of the final product before being offered on the market. There are also many other features but the last one is the one we are considering for our tool. As industries increasingly embrace automation to enhance efficiency and streamline operations, the need to optimize AGV movement and traffic flow becomes paramount. Smart addresses this need by providing a comprehensive simulation environment that allows plant managers, engineers, and operators to visualize and optimize AGV traffic scenarios.

Key features: Smart creates a lifelike representation of the manufacturing plant, including

pathways, work zones, intersections, and loading/unloading stations. The software uses advanced graphics to emulate the plant's layout, aiding in visualizing AGV movements. Users can define various traffic scenarios by specifying the number of AGVs, their routes, destinations, and tasks. This enables testing and optimizing different traffic configurations before implementation. Smart provides detailed insights into traffic patterns, bottlenecks, and congestion points. This information is crucial for identifying areas where AGV movement can be improved to enhance operational efficiency. The software allows users to schedule tasks for AGVs, such as material transport, product delivery, or equipment handling. Users can assess how different task schedules impact AGV traffic and overall plant productivity. Users can adjust parameters like AGV speed, acceleration, and braking behavior. This flexibility helps analyze the effects of various parameters on traffic flow and system performance. Smart offers real-time monitoring of AGV movements, displaying their positions, routes, and task status. This live view facilitates tracking and analysis during simulation runs. The software provides performance metrics, such as average travel time, utilization rate, and task completion time. These metrics help evaluate the effectiveness of AGV traffic management strategies.

There are also many benefits to having a simulation tool:

Smart empowers plant managers to optimize AGV traffic flow, reducing congestion, wait times, and resource wastage. The ability to test different scenarios virtually minimizes the risks associated with implementing changes in the physical plant. By identifying traffic bottlenecks and inefficiencies, Smart helps enhance overall plant productivity and operational performance. Smart can also serve as a training tool for AGV operators and a platform for analyzing AGV behavior under various conditions.

In conclusion, Smart offers a valuable solution for manufacturers seeking to enhance AGV traffic management within their plants. By providing a realistic simulation environment, the software aids in optimizing AGV movements, reducing operational challenges, and ultimately contributing to improved productivity and efficiency in manufacturing operations.

In summary, the features we are planning to test in relation to Smart include the correct opening and execution of a layout file, the correct order of task completion given the same input and a time performance control to exclude the presence of deadlocks or temporary blocking stages during the simulation. As smart is provided with an API we will be able to communicate with it through designed endpoints.

4 | Background

Before we move over to the design of our project, we'll have a quick look at the available technologies that we evaluated but decided not to adopt. Introducing pros and cons of these technologies requires to make a distinction between Unit Testing and Functional Testing, taking into account what are our prerequisites and our goals.

4.1. Unit Testing

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually scrutinized for proper operation. Software developers and sometimes QA staff complete unit tests during the development process. The main objective of unit testing is to verify that each individual unit behaves as expected and produces the correct output given a set of inputs.

There are many advantages to unit testing, including the following:

1. Automation: Unit tests can be performed automatically, which means they can be run repeatedly and consistently whenever needed.

2. Granularity: Each unit test targets a specific aspect of the unit's behavior. This fine granularity ensures that defects can be localized and addressed more effectively. This will help the developer to act on the codebase more efficiently.

3. Early Detection: Unit testing finds problems early in the development cycle. This includes both bugs in the programmer's implementation and flaws or missing parts of the specification for the unit. The process of writing a thorough set of tests forces the author to think through inputs, outputs, and error conditions, and thus more crisply define the unit's desired behavior.

4. Reduced Costs: The cost of finding a bug before coding begins or when the code is first written is considerably lower than the cost of detecting, identifying, and correcting

the bug later.

5. Regression Testing: Unit tests act as a safety net against regressions. When code changes are made, running unit tests helps ensure that existing functionality remains intact.

6. Documentation: Well-written unit tests serve as a form of documentation for how a particular unit should behave. Developers can refer to tests to understand the expected behavior of a unit.

7. Refactoring: Unit tests support code refactoring by providing confidence that changes won't inadvertently break functionality. If tests pass after refactoring, it indicates that the code still functions correctly.

8. Continuous Integration: Unit tests are often integrated into a continuous integration (CI) process, where they are automatically executed whenever new code is pushed to a shared repository. This ensures that changes don't negatively impact the existing codebase.

9. Test-Driven Development (TDD): TDD is a development approach where developers write unit tests before writing the actual code. This practice encourages thinking about the desired behavior before implementation.

Although it has many strong points, unit testing comes with some disadvantages as they may not cover every type of bug. Integration bugs won't be caught as the single unit is being testes alone or, for example, a single function may not be tested against every possible input that could show up after release. Single-line function in the codebase may require multiple and complex lines of code, creating a potential time investment.

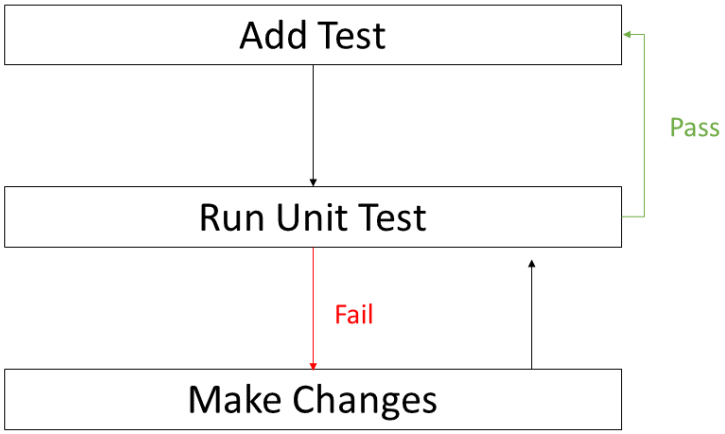

Figure 4.1: Unit Testing Steps

4.2. Functional Testing

Functional testing is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Each function is compared to the corresponding requirement to ascertain whether its output is consistent with the end user's expectations. The testing is done by providing sample inputs, capturing resulting outputs, and verifying that actual outputs are the same as expected outputs.

There are some benefits from functional testing, including the following:

1. Requirement Validation: Functional testing ensures that the software aligns with the specified requirements, validating that it meets user expectations and business needs.

2. End-to-End Testing: Functional testing often includes end-to-end scenarios, simulating real-world user interactions, which helps uncover issues that might arise during actual usage.

3. Bugs Detection: Functional testing detects issues related to missing or incorrect functionality, such as incorrect calculations, data inaccuracies, or faulty logic.

4. Regression Testing: As the software evolves, functional testing aids in ensuring that

new features or changes do not adversely affect existing functionality.

5. Validation of Integrations: During integration testing, functional testing verifies that different components or modules work cohesively when integrated into the complete system.

6. Verification of Business Logic: Functional testing validates that the software correctly implements business rules, ensuring accurate processing of data and transactions.

Along with its upsides, functional testing comes along with its downsides:

1. Limited Scope: Functional testing may not cover every possible scenario, leaving room for defects that occur under specific conditions or edge cases.

2. Time-Consuming: As software grows complex, the number of scenarios to test increases, potentially making functional testing time-consuming.

3. Narrow Focus: Functional testing might emphasize functional correctness at the expense of other aspects like performance, security, or usability.

4. Incomplete Coverage: Comprehensive functional testing requires extensive test cases, which can be difficult to achieve in large or intricate applications.

5. Dependency on Documentation: Functional testing relies heavily on accurate and up-to-date documentation, making it less effective if documentation is lacking.

6. High Maintenance: As the software evolves, functional tests need to be updated to reflect changes in requirements, potentially leading to maintenance challenges.

7. Static Testing: Functional testing typically doesn't evaluate how the software performs under dynamic conditions, such as load, stress, or real-world network fluctuations.

8. Not Always User-Centric: While functional testing ensures that the software meets requirements, it might not identify issues that impact the user experience negatively.

In conclusion, functional testing plays a crucial role in ensuring that software behaves as intended and meets user expectations. However, it has limitations in terms of coverage and scope, and it's important to supplement it with other testing methods to achieve a comprehensive understanding of the software's overall quality.

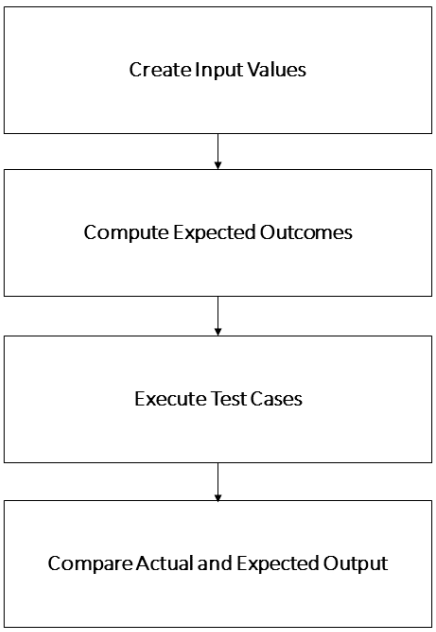

Figure 4.2: Functional Testing Steps

4.3. Prerequisites and Goals

Since i started this project i was introduced to what are Smart and SmartCad, and to which functionalities of the two software we wanted to focus on.
As SmartCad is in charge of lots of things it would be nearly impossible to test all of them so we had to narrow it down to what are the main aspects of the program. Comparing layout files, for example, requires a tool able to read database files and also a user-dependent configuration to decide what, in terms of tables or fields, has to be compared and what can be skipped to speed up time computation. Comparing export files instead requires a tool being able to read plain text files or binary files, with an ad-hoc solution to know, given the version, which metadata must be skipped and where it is situated inside the file. Time performance is also an issue as the generation of a correct output cannot overwhelm a time generation that's double the time of a test case's.
On the other hand Smart, being in charge of real-time simulation, is quite challenging to test. Static and ideal path planning can always be easily tested by using some JSON files to save the returned output whilst real-time simulation requires a bit more complex algorithms to check that the same operations are executed in the same (or almost) order and in a reasonable amount of time.
Given this information and the above upsides and downsides of both Unit and Functional Testing we decided to proceed with the second option.

Unit testing has some very strong points but lacks of two essential features: integration, which is essential in software like these, and the fact that these software where not born with a Test-Driven Development idea, meaning that an enormous multitude of tests should have been created and with the remote chance that they could lead to a relevant codebase modification.
Functional testing, instead, will overshadow the fact that the two software lacks the TDD structure and certainly guarantees the integration of all the involved modules. There are some downsides that will limit our tool but there also some workarounds that can help us to diminish these unintended drawbacks. We enlarged the narrow focus issue by checking the performances during both output and test case generation, documentation is and will be kept up-to-date and all the related issues involving user and coverage might be resolved quickly as the end users are the actual team members that know which new functionalities have been introduced and what input could have caused an anomaly.

4.4. Sqldiff

Looking for some open source software for database comparison we came across Sqldiff [7], which seemed the most promising among the available software. The Sqldiff utility works by finding rows in the source and destination that are logical "pairs". The default behavior is to treat two rows as pairs if they are in tables with the same name and they have the same rowId, or in the case of a WITHOUT ROWID table if they have the same PRIMARY KEY. Any differences in the content of paired rows are output as UPDATEs. Rows in the source database that could not be paired are output as DELETEs. Rows in the destination database that could not be paired are output as INSERTs.
Being a command line program it was convenient to adopt as a graphical interface would have caused the automated part of our tool to fall apart. Nonetheless it couldn't be adopted because of their interpretation of "pairs": layout files may ,as intended, change from one version to the next one, leading to the same table having a different number of rows (due to more rigorous or flexible calculations) or it could even have the same number of rows but they might be shuffled. This would cause the trigger of a false negative, invalidating the test.
We therefore had to create a solution ourselves and in the next chapters we will explain how we treated this scenario.

4.5. WinMerge

With a quick look for some software to compare the differences we came across Winmerge [6].
WinMerge is an open-source differencing and merging tool for Windows, designed to help users compare and merge text files and directories. It is particularly useful for software developers, content creators, and anyone who needs to track changes in files, folders, or codebases. WinMerge provides a user-friendly graphical interface that facilitates visual comparison and merging of files and directories, allowing users to identify differences and synchronize content easily. Winmerge is also equipped with an option to input files in an inline manner. The problem with this CLI version of the software is that the output would be either shown via graphical interface or saved as a HTML document. As we would still have to read a file and due to the fact that txt and bin files' reading is quite trivial we decided to find a solution on our own.

5 | Related Work

This chapter presents an overview of the related work in the field of automated software testing, with a specific focus on functional testing. The exploration encompasses various methodologies, techniques, and tools employed for automating the testing process. By examining prior research and industry practices, this chapter lays the foundation for the current study on automated software testing through functional testing.

5.1. Introduction to Software Testing

Before we started coding we wanted to have a solid foundation to understand the principles of software testing and its critical role in software development. The book [8] presents a systematic approach to software testing that encompasses both theoretical knowledge and practical techniques. It introduces the fundamental concepts of software testing, emphasizing the importance of testing in ensuring software quality and reliability. It outlines various testing techniques, including white-box testing, black-box testing, and gray-box testing. They explain how these techniques can be applied to uncover different types of defects and vulnerabilities in software.The authors also provide insights into the process of designing effective test cases. They discuss strategies for generating test cases that target specific aspects of software functionality and behavior. It addresses the role of automated testing in modern software development, discussing the benefits and challenges associated with automated testing tools and techniques. Overall the book is a comprehensive resource that equips readers with a thorough understanding of software testing principles, methodologies, and techniques. The book's combination of theoretical insights and practical examples makes it an essential reference for anyone involved in software development and testing.

5.2. Black Box vs White Box Testing

White box testing [11] is a form of application testing that provides the tester with complete knowledge of the application being tested, including access to source code and design

documents. This in-depth visibility makes it possible for white box testing to identify issues that are invisible to gray and black box testing. The most renowned techniques are `statement coverage testing` that ensures that every line of code within an application is tested by at least one test case. `Statement coverage testing` which can help to identify if portions of the code are unused or unreachable, probably caused by programming errors, updates, etc. Identifying this dead code enables developers to fix incorrect conditional statements or remove redundant code to improve application performance and security. `Branch Coverage` where conditional statements create branches within an application's execution code as different inputs can follow different execution paths. Branch coverage testing ensures that every branch within an application is covered by unit testing. This ensures that even little-used code paths are properly validated. In `Path Coverage` an execution path describes the sequence of instructions that can be executed from when an application starts to where it terminates. Path coverage testing ensures that every execution path through an application is covered by use cases. This can help to ensure that all execution paths are functional, efficient, and necessary.
Black Box Testing [9] is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing. The most renowned techniques are `Equivalence Class Testing` which is used to minimize the number of possible test cases to an optimum level while maintaining reasonable test coverage. `Boundary Value Testing` focuses on the values at boundaries, this technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges. There are many types of Black Box Testing but the one we will focus on is functional testing – This black box testing type is related to the functional requirements of a system; it is done by software testers.
Even if a detailed comparison can be made between the two [10] we opted for a black box testing approach. We were not gained access to the whole of the source code but just to some available API endpoints and to some method names to call through reflection using just some DLLs. As these factors invalidate the use of a white box testing approach we went for the black box testing approach, specifically with the functional testing one even though many testing types were available.

5.3. The Importance of Functional Testing

In the field of software engineering and quality assurance, functional testing refers to the process of evaluating a software application's functionality to ensure it performs as expected and meets the specified business requirements. A perspective from business practice would likely emphasize the importance of functional testing in delivering a reliable and user-friendly software product that aligns with the organization's goals and objectives.
In this article [13] it is highlighted the critical role that functional testing plays in identifying and addressing software defects, thereby reducing the risk of system failures and customer dissatisfaction. It should be closely aligned with the specific business goals and requirements of the software project. This alignment ensures that the software serves its intended purpose and supports the organization's objectives. It introduces the user to various strategies and methodologies for conducting functional testing, such as manual testing, automated testing, and exploratory testing. The pros and cons of each approach and when to use them are taken into consideration. It also indicates the importance of clear reporting and documentation practices in the context of business projects and considers the cost-effectiveness of testing efforts and resource allocation.
Given this premises we were decided to embrace the family of functional testing and to move to the practical aspect of it. There were three choices at the beginning: one was to buy an already existing tool, use a open-source software or to create our own ad hoc solution. The first solution was a bit ineffective as being cost-efficient is one of the strength of functional testing. The choice definitively fell on the last two solutions.

5.4. Why not use open-source functional testing tools?

The primary focus of this article [12] is on assisting software professionals in making informed decisions when selecting testing tools to improve the efficiency and effectiveness of their testing processes. The authors emphasize the growing importance of testing tools in the software development process. It presents a systematic approach for evaluating and selecting testing tools. A set of criteria that organizations should consider is outlined when assessing potential testing tools. These criteria may include functionality, ease of use, compatibility with existing systems, cost, support, and scalability. Testing tools are therefore classified into different categories based on their intended purposes, such as test case generation, test execution, test management, and defect tracking. They provide guidance on which types of tools are suitable for various testing tasks. It stresses the importance of integrating testing tools into the software development process seam-

lessly. Tools should align with the organization's software development methodologies and practices. Conducting a cost-benefit analysis when selecting testing tools is discussed throughout the article, as organizations should weigh the costs of acquiring and implementing the tool against the potential benefits it can bring in terms of improved testing efficiency and higher software quality. Vendor support and the availability of updates and maintenance are essential considerations in tool selection. The article also offers guidance on how to assess the reliability and support offered by tool vendors.
Using open-source functional testing tools offers numerous advantages, but it's important to recognize that they may not be the best fit for every project or organization. These open-source software are certainly backed by a supportive community and they surely are cost effective, nevertheless they may not fit the company's toolchain requiring some integration effort. Many popular open-source tools are Selenium, Cypress and Playwright, however these won't fit the requirements of our purpose as they, and the majority of functional testing software, were created to stress web related software and to record which sequence lead to a fault in the system. Since our tool is not web related and that there already exists a way to know what caused a problem to occur, we can avoid adopting open-source tools so that no integration or fix on updates will be needed.

6 | Design

6.1. High Level Architecture

The application architecture consists of two distinct front ends, a Command-Line Interface (CLI) and a Graphical User Interface (GUI), both of which share a common back end. This design allows users to interact with the application using their preferred interface, whether it be through the command-line for more streamlined operations or a graphical interface for a more user-friendly experience.

The back end of the application serves as the core processing engine, handling the business logic and data manipulation. It operates on two different company software systems, which means it can communicate with and extract data from these separate software applications, leveraging their functionalities while maintaining a cohesive user experience.

One of the key tasks of the back end is to access and manage various data sources. This includes working with database files, logs, and other system files associated with the two company software systems. The back end is responsible for reading, writing, and updating data in these files to facilitate seamless data integration and processing across the application.

The CLI front end offers a text-based interface that allows users to input commands and parameters directly into the terminal or command prompt. This interface is often favored by power users, administrators, or developers who prefer a command-driven approach for rapid execution and automation of tasks.

On the other hand, the GUI front end provides a visual interface with interactive controls, menus, and graphical elements that enhance user interaction. This interface is designed to be more intuitive and user-friendly, making it suitable for less technically inclined users or those who prefer a point-and-click approach.

Regardless of the front end used, both the CLI and GUI communicate with the shared back end. This ensures consistency and avoids duplicating code, allowing the application to be more maintainable and efficient.

The application's back end is designed to be versatile and adaptable, making it capable of seamlessly integrating with different company software systems. It abstracts the complexities of interacting with these systems, providing a unified and coherent interface for both front ends to interact with the various software components.

In summary, the application's architecture with two front ends (CLI and GUI) and a common back end allows users to interact with the same powerful processing engine in their preferred manner. The back end works with two different company software systems, manages data from multiple sources, and facilitates a smooth and efficient operation across the entire application. Whether users prefer the simplicity and speed of the CLI or the user-friendly interface of the GUI, the back end ensures that data and functionalities are effectively shared and utilized across both front ends.

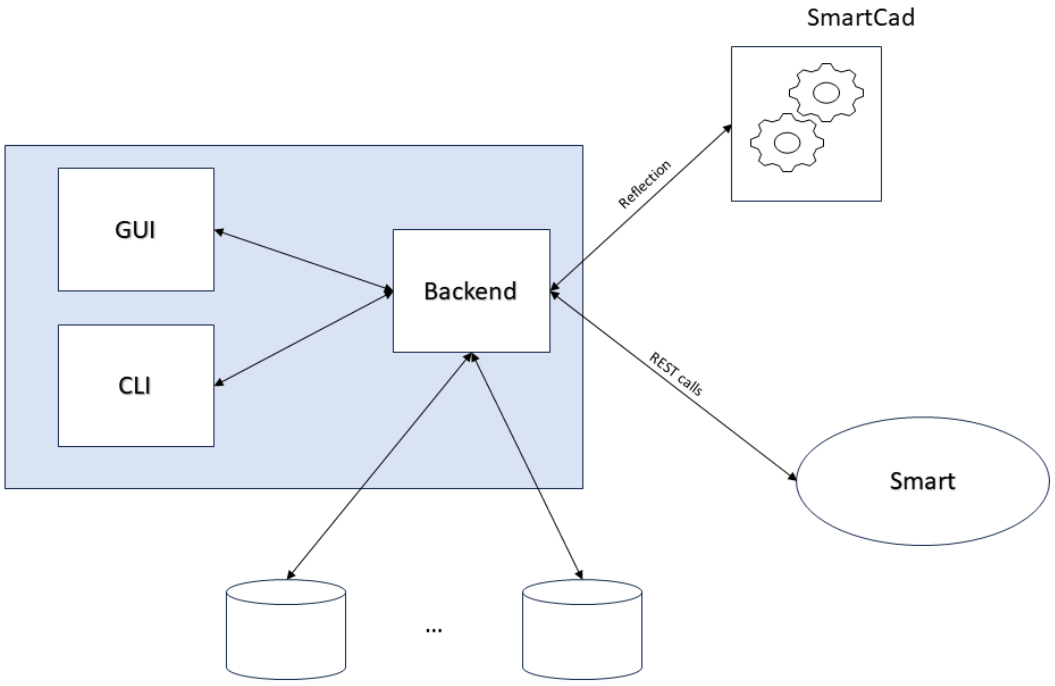

Figure 6.1: High Level Architecture

6.2. Back end design

The back end of the application serves as the central processing engine that orchestrates various tasks related to test case generation, data comparison, and simulation of factory traffic. It is a critical component responsible for coordinating the interaction between

different software systems, generating test cases, executing simulations, and reporting errors.

Here is a more detailed description of the back end's functionalities:

1. Test Case Management: - The back end maintains a repository of available test cases and understands which test cases need to be generated to cover specific scenarios. It may use configuration files or databases to keep track of test case information.

2. Dynamic DLL Loading and Test Case Generation: - The back end utilizes reflection to dynamically load DLLs from one of the company software systems. This allows it to access functionalities and methods present in these DLLs to generate additional test cases that are not explicitly defined.

3. Data Comparison: - After generating the necessary test cases, the back end compares the output files, including database files and binary files, from the different company software systems. It checks for any differences or discrepancies in the data to ensure consistency and accuracy.

4. Simulation of Factory Traffic: - The back end interacts with Smart to simulate the traffic in a specific factory environment. It achieves this by making REST API calls to the software, which in turn responds with relevant data and status updates. In this way the back end acts as a REST API client, making the necessary HTTP requests and handling the responses.

5. Error Reporting: - Any errors or exceptions that occur during the test case generation, data comparison, or factory simulation process are logged and reported to an XML file. This file acts as a comprehensive error log, containing information about the nature and location of the errors.

6. Logging and Debugging: - The back end may also incorporate logging mechanisms to record the execution flow and important events during the testing and simulation processes. This helps in debugging and understanding the application's behavior.

7. Data Handling and Transformation: - The back end processes and transforms the data obtained from different sources, preparing it for comparison and further analysis.

Overall, the back end acts as the brains of the application, intelligently managing test cases, calling external DLLs for additional test case generation, comparing data, and orchestrating factory simulations. Its role in error reporting and logging ensures that any issues or discrepancies are promptly identified and documented for further investigation. The back end's flexibility, facilitated by reflection and REST API calls, allows the ap-

plication to adapt to different scenarios and software systems, making it a powerful and versatile tool for ensuring software compatibility and factory simulation.

6.3. Configuration

In modern software development, the method of efficiently storing and managing configuration settings significantly impacts the flexibility of applications. A prevailing strategy involves employing JSON (JavaScript Object Notation) files as a robust means for configuration storage. JSON, known for its lightweight and human-readable format, proves itself as an ideal candidate for housing various software configuration parameters.
As we opted for an ad-hoc solution we wanted our tool to be easy to configure and flexible, so that we can even be able to test just a restricted part of the functionalities that the software can overview. Our configurations include some fields to store directory paths, allowing users to save some of the folders that we will show in the next chapter wherever they want, some boolean flags to decide whether some functionalities have to be tested or not and some parameters to choose, within a functionality, if something should not be compared. The best way to allow this degree of freedom was to utilize a JSON file. Given that it supports a large number of data types and its serialization and deserialization are well renowned and optimal it was perfect to store many different types of information. Including that as both company software evolve our tool will probably evolve with them, making it easy to add more configurations. Applying this configurations is quite trivial as, thanks to its human readability, the JSON file can be easily modified by hand without the need to make changes to the code. Another way of modifying the configurations has been introduced in our GUI, making available an user-friendly method. This will be later introduced in the next chapter.

6.4. Folder Structure

The overall idea is to test the candidate release version against the results given by at least one older, but stable, version of the same program. To fulfill this requirement we will have one folder for storing the latest version and one for storing the older ones.
If the software we are going to test is SmartCad then will have to firstly have a quick check to assure that all the requested libraries are contained in each folder, we can then continue by checking if in a third folder the test case for that specific version and that layout already exists. If it does then the first part of our program is skipped for that file, not creating any new file, otherwise a new layout will be generated and 2 new files will be created.

Otherwise, if the software we are going to test is Smart then four folders need to be available. One for storing the latest Smart version, one for storing the old Smart version, one to contain all the layouts we are going to test and a final one for storing the generated test cases.

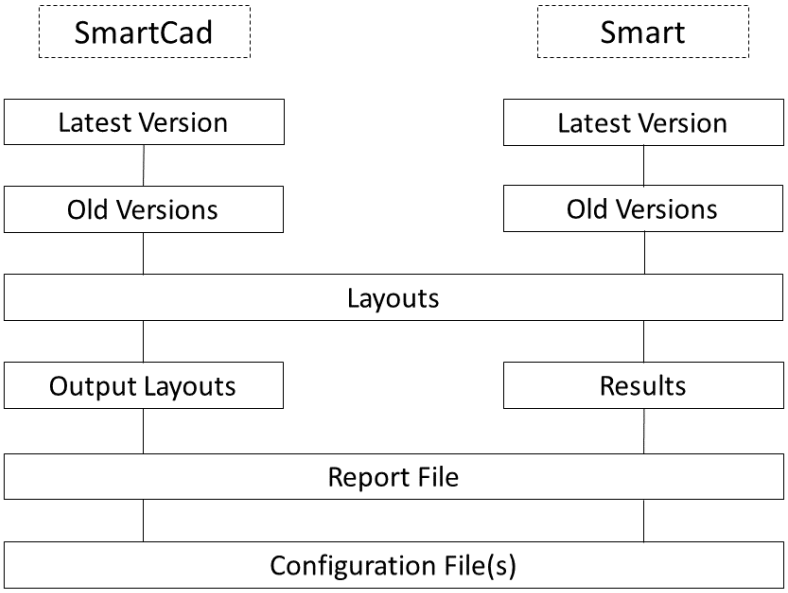

Figure 6.2: Folder Structure

In figure 6.2 it is possible to observe that the folder Layouts is shared between the two, making the structure less redundant and less prone to inattention errors. The Layouts folder contains all the stable layouts (meaning they have generated with a stable version of SmartCad, the ones contained in the Old Versions folder) that our tool has generated up to that moment, including some that can be manually saved by some user, all grouped by layout name and SmartCad version. If perhaps a certain layout has already been saved with a specific SmartCad version then it would be futile and a waste of time to re-generate it, else if that file does not exist then it would be freshly generated and saved in the Layout Folder. The Layout folder also contains some time performance data for each file that has been generated, assuring there are no big differences in time steps. While SmartCad has an Output Layouts folder to temporarily stash all the layouts and time performance data generated from the SmartCad Version contained in the Latest Version folder, Smart has a Results folder to save data generated through the overview of simulations also grouped by name and Smart version. The Report file is created in such a way that it can represent

both Smart and SmartCad outputs, avoiding redundancy. More than one configuration file can be saved, enabling the user to easily switch between them before launching the test both from CLI and from GUI.

6.5. Blending SmartCad and Smart

Since this moment we have just illustrated Smart and SmartCad as standalone software, illustrating their requirements, their feature and their goals. Now we are going to illustrate roughly how they blend in with each other to further understand how our tests will be structured.

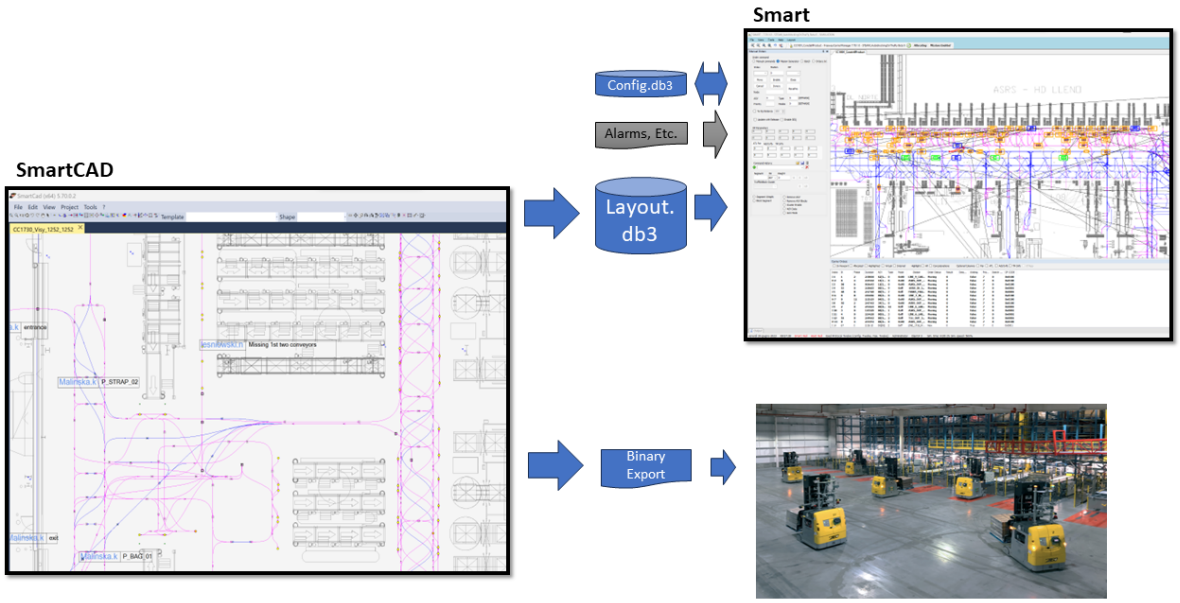

Figure 6.3: SmartCad and Smart Blending

SmartCad will draw efficient AGV trajectories considering speed, geometries and Safety Areas, it will set traffic rules as blocks, escapes, home positions, clusters, weights, it will export the binary files required by the AGVs and many other things. Most of this information is saved in a Layout.db3 as image 6.3 suggests. This file, along with other configuration and alarm files, is then loaded by Smart which can be either used to control real AGVs movement inside a plant or it can be used in simulation mode to test the AGVs' behaviour.
By looking at figure 6.3 it seems easy to understand that when we are testing SmartCad, creating a new Layout.db3, we also have to test that the output can be loaded by Smart

and that the simulation runs smoothly. On the other side the opposite is not true: testing a new Smart version requires just one stable Layout.

6.6. SmartCad Tester Design

Here we are going to quickly illustrate what are the major steps in the SmartCad Test, while in the next chapter [Ch.7] we will focus on which where the major issues and how we resolved them. This workflow is summarized in figure 6.4.

1. Starting from the Layouts' folder we select one Layout.db3 per plant and evaluate all the possible permutations of that plant with the available stable SmartCad versions.

2. This stage is optional as all permutations of the test cases may have already been produced in a previous test. If that is not the case then they will be generated and saved in the Layouts' folder.

3. The latest version layout will be generated and all its data will be saved in the Output Layouts' folder.

4. The latest layout file will be compared to all the stable layouts file, reporting differences for each of them.

5. The latest export files will be compared to all the stable export files, reporting differences for each of them.

6. The same Smart version will be used to run one Smart simulation for one stable layout file and one will be launched for the latest layout file. Times and task orders will be saved and compared. In the next section 6.7 we can have a look at the design of the Smart Tester.

If all the available plants have been tested the test ends otherwise it loops back to point number 1, selecting a new plant.

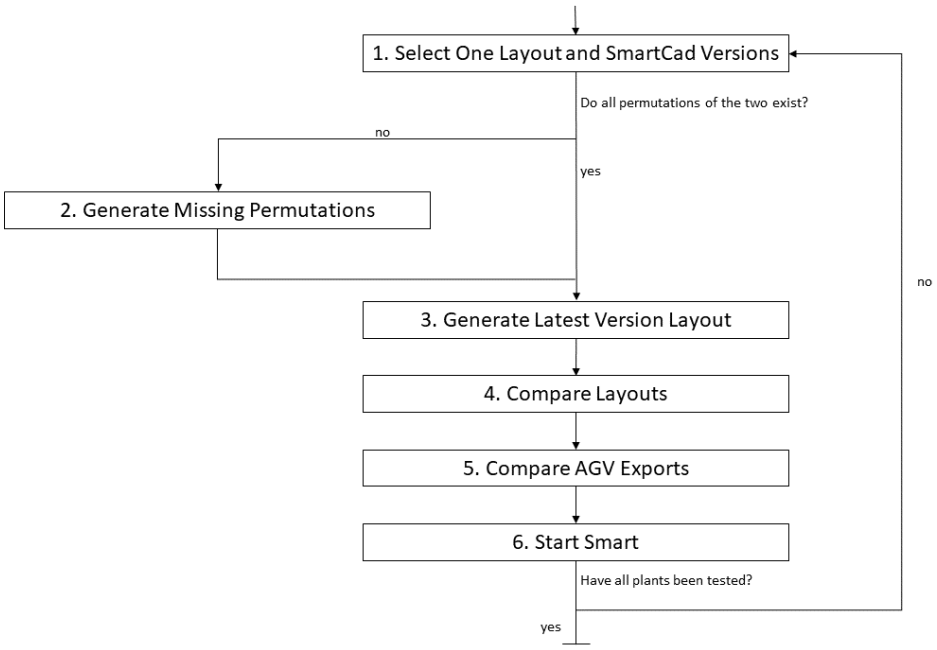

Figure 6.4: SmartCad Tester Workflow

6.7. Smart Tester Design

The figure below 6.7 illustrates what the main steps in the Smart test are. As we mentioned before (section 6.5) Smart can be either tested as an independent software or as a second test after the SmartCad one. This will have minor changes in point 1 and 6, but the gist of it is still the same.

1. Here we retrieve the layout file and the Smart version to be tested, whether it's the first one or the second one it does not matter as we are going to retrieve just the stable version for both of them. We further check if the results for this permutation have already been computed or not, so that we know if we can skip to point 6 or we have to start Block1's execution.

2. If no result has been found then we will make a process call to Run Smart with the given layout.

3. We then compute some random static, or ideal, paths with all the data they furnish back.

4. We start a simulation, taking notes of times and tasks orders.

5. Results are saved in a JSON file.

6. If we are testing Smart as a standalone software then we are going to keep the same layout and swap just the Smart version to the latest available, otherwise we are going to switch layout to the freshly generated one and use the stable version of Smart.

7. We are going to execute the Block1 steps (which consists in points 2, 3, 4 and 5) for this permutation, using the same start and end points for static path computation and the same orders for the simulation.

8. The results have been saved, compared and shown to video.

If all the available plants have been tested the test ends otherwise it loops back to point number 1, selecting a new plant.

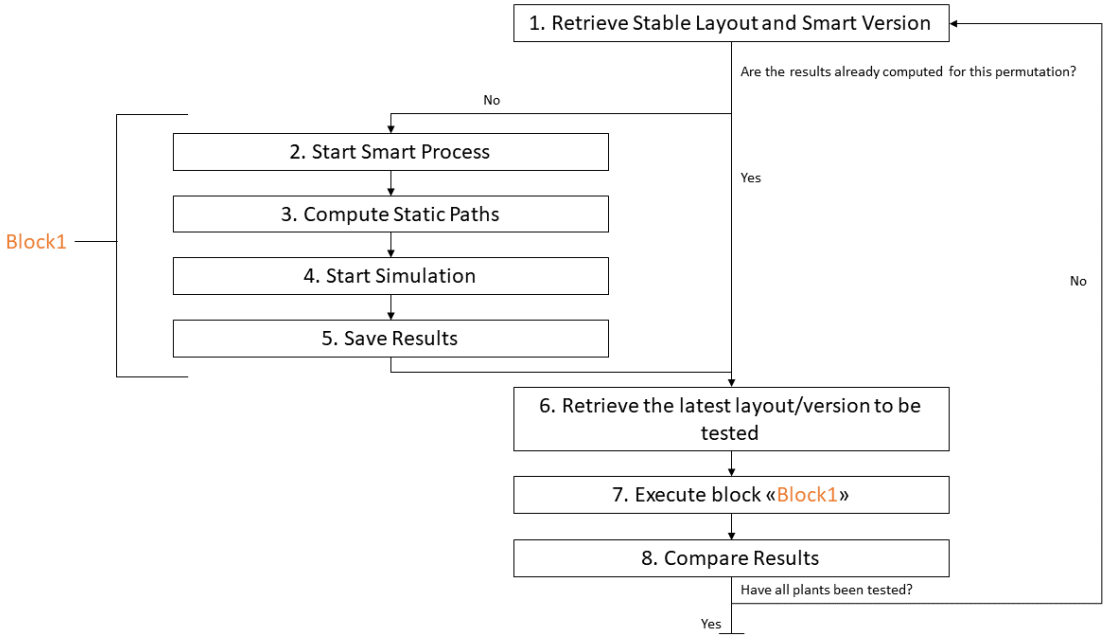

Figure 6.5: Smart Tester Workflow

7 | Implementation

In this chapter we will analyze some practical aspects of the actual implementation of the system. Starting from the operations we performed on the SmartCad tester and moving on to the Smart tester. Last but not least we will show the final GUI and CLI, illustrating how settings can be modified and how to interact with the system.

7.1. Layout Generation

Since our SmartCad tester could generate many layouts depending on the version and due to the fact that we don't want to hard code the necessary DLLs into the software. Every new SmartCad version that has been approved to be offered on the market would then have the outcome of re-compiling the tool. To avoid this manual and inefficient job, we are going to dynamically load the required assemblies taking them from their respective SmartCad version. This is achieved through reflection, which refers to a programming technique that enables a program to examine and interact with the contents and structures of a DLL at runtime. Reflection allows an application to access metadata, types, methods, properties, and other members within a DLL without having prior knowledge of their definitions during compile-time. Instantiating a new AssemblyLoadContext will allow us to create a scope for loading, resolving, and potentially unloading a set of assemblies.

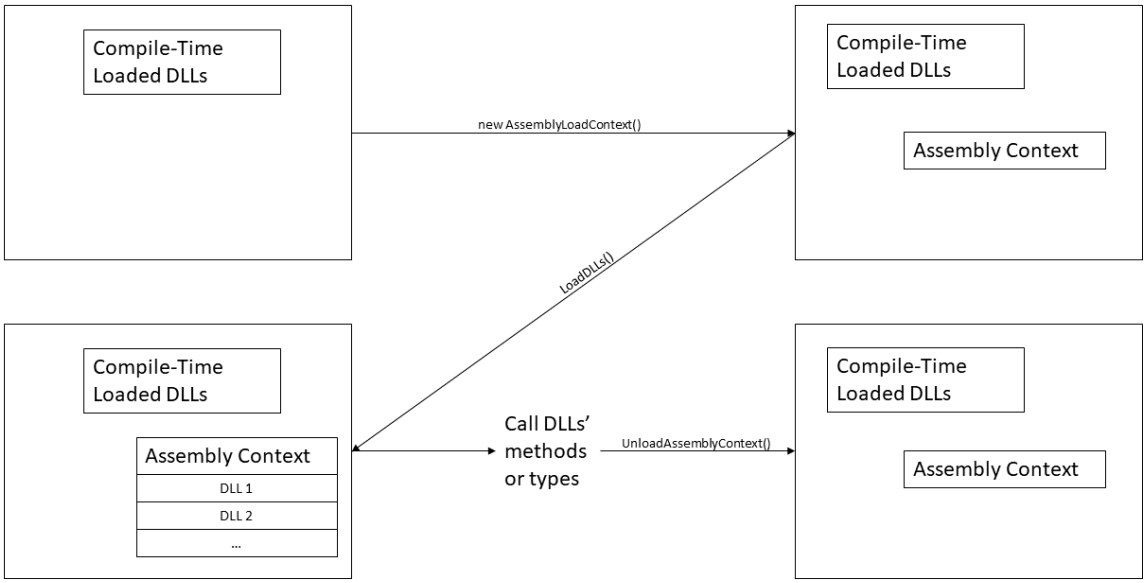

Figure 7.1: Assembly Loading Context

Therefore we are going to load all the required assemblies in our AssemblyLoadContext, then we proceed to the reflection of the SmartCad methods and types to create a new layout file. Please notice that all the types that have been used in the portion of the code are 'dynamic type', as the program has no information of their types at compile time. This made debugging a bit complex. Firstly we fetch the SmartCad version of the layout that we are observing. The second step is to create a serializer that handles the loading and saving procedures. Feeding the serializer with the layout and the layout's version allows us to call the method `OpenLayout()`, to correctly open the layout with the proper deserialization function. Then `GenerateAutoBlocking()` and `ExecuteAGVExport()` are two methods that are not mandatory and, as we precedently said in the configuration section [6.3], they can be bypassed if the user wants to. Their exclusion from the test may be done because the autoblock's generation can be very inefficient as some layouts may contain more than 4 million records in that single table. We then re-uuse the serializer to call the method `SaveAs()` to save the layout with the SmartCad version that we desire. Ultimately we unload the assembly context to be prepared in case new DLLs have to be loaded.

7.2. Comparing Layouts

When we talked about why we decided not to use SqlDiff (section [4.4]) we said that their idea of pairs did not fit our idea of pairs. So let me introduce you to our idea of equal: two records in a database table are considered equal when all their corresponding attributes or fields have the same values. In other words, if every piece of data in both records is identical, then those records are considered equal. Therefore if two records are equal then they form a pair. Unfortunately another problem arose: the multitude of records that each table may have, including that there are more than 50 tables with some containing more than 3 million records. Copying this tables into a `DataSet` [1] is quite trivial but how to make the comparison wasn't. Another issue was that in some tables some values were repeated and there was not a primary key, making a compare just on the primary was not feasible so we had to find a more general approach. A first approach consisted in ordering the two DataSets, start moving between the two using two auxiliary variables (i and j) and gradually compare every field of a row. If the two rows were equal then both i and j would have had proceeded, otherwise just j would have had proceeded until a match was found or the last row had been evaluated (making i proceed and restoring j back to the last equal value plus one). This approach was too slow because if the latest layout added 10 new rows at the beginning of a table then there would have been `O(10 * N)` futile checks.
A second idea was to use not just "equals" as a math comparer but also less and greater. Using i and j whilst checking the Datasets the first would have had proceeded if its current field was less than j's and vice versa. If all the fields in a row were equal then they would have had made a pair. It wasn't that bad but we noticed that if in the latest SmartCad version a field changed its default value from "" to "empty" all the rows would be considered different, making `O(2 * N)` futile checks in the worst case.
The last solution we had consisted in skipping the ordering part and create for each row its own `HashSet` [3], which in turn is created from the HashSet of all its fields. The lookup operation is `O(1)` for each element so we are looking for a `O(N)`. This has proved to be the most efficient way to compare two DataSets.

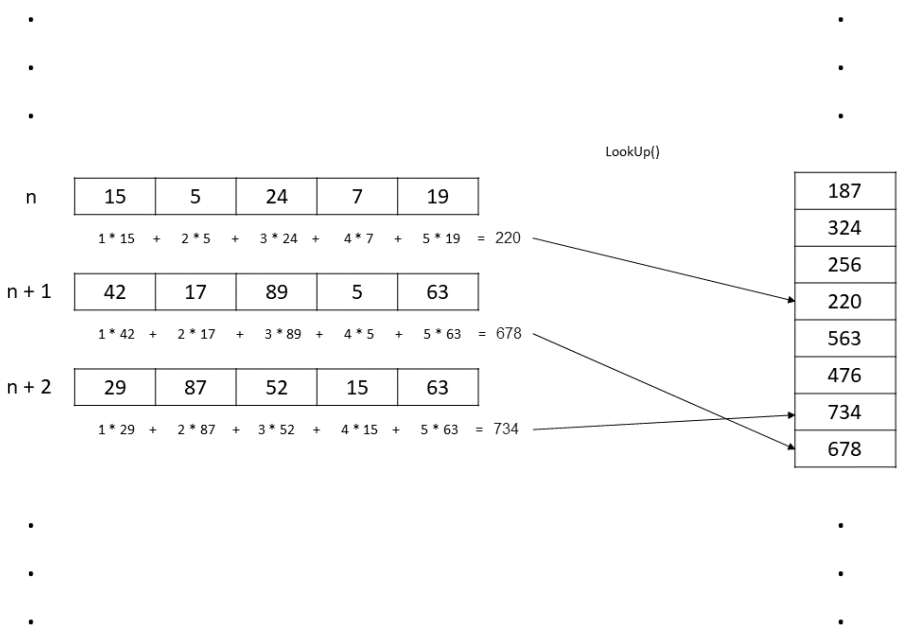

Figure 7.2: HashSet

7.3. Multitasking

We have said many times that our layout files are relevantly big and that they contain a moderate number of tables. Given that one of the requirements for this tool is to being able to give a response as quick as possible (see section 2.2) then we opted for a multitasking code execution to speed up computational time while waiting for something substantial to execute. The first thing that had to be run on its own separate `Task` [5] was the GUI. Every time the test was launched it would freeze, disabling the users to minimize or even closing the window. Moving on, creating our DataSet from reading the layout file through SQL calls is time consuming. The DataSet structure, however, doesn't allow concurrent writing so we where limited to insert one `DataTable` [2] at a time. SQLite however allows concurrent reading from one file, so our strategy was to spawn a number of task equal to the number of tables, start asynchronously reading all the tables and once one was ready it was added into the DataSet. Comparing the whole DataSet was also an issue, so the same strategy was applied: the same number of tasks is spawned and each one of them is in charge of calling the `CompareDataTable()` method, which utilizes the strategy described before in section [7.2].

7.4. Simulation

Once a Smart process has been created we can interact with it through an API to yield the required information that we need. Here is a list of the main endpoints available on the RESTful API service.

7.4.1. /

ACTIONS: GET
This is the first trivial endpoint that we come across. It is just used to verify that the service is properly running.

7.4.2. /carrier-manager/routing-tool/path

ACTIONS: GET
Before we start simulating we need to retrieve what are the ideal paths that our agvs would use to travel from point A to point B. We randomly generate a number of points, then we look for the path from one point to its successor. Many information are yielded by this endpoint as all the evaluated costs, the distance, the orientation of the vehicle and some other information that shall not change.

7.4.3. /simulator/speed

ACTIONS: GET - POST Setting the speed is quite relevant for the correct execution of the simulation. In a simulation we can move faster than in real life, avoiding wasting time. But how fast can we go? It depends, we can't go too much fast as then some or all traffic rules could be broken whilst we can't even go too much slow as we want our test to be as fast as possible. The top level speed depends on many facotrs such as number of vehicles in the plant, the number and the disposition of the segments, clusters and many other things. As the layout has a relevant role in the speed we can set we must assure it coincides with the one saved in a configuration file and if that does not happen then we set it to that value.

7.4.4. /traffic-manager/orders-from-file

ACTIONS: POST The preconditions to make this API call is to have saved a file in the layout's folder inside Smart. This file can be generated from an old version of Smart and it can be reused forever. It contains the orders that we want to be completed in specific

format to specify starting and ending points, along with some intermediate points, priority, the AGVs types required to make that order and other info. Once the call is done the simulation starts taking the orders from the file.

7.4.5. /carrier-manager/state

ACTIONS: GET This is the main endpoint used to poll the state of simulation including the AGVs position, current orders, alarms, etc. We basically continuously poll the API to know if issues were raised and to know when an order has been completed so that it can be added to the list of completed orders. This list won't just contain orders but it will also contain other information as the id of the AGV that took charge of the order, the distance it made and the time it took to complete the order. These info will be used in the tester when an order takes more time than it should. This list will be saved in a JSON file to avoid repeating the simulation if it's not necessary.

7.5. GUI

This is the graphic user interface of the program which allows to easily modify the configuration file and to run both the tests.

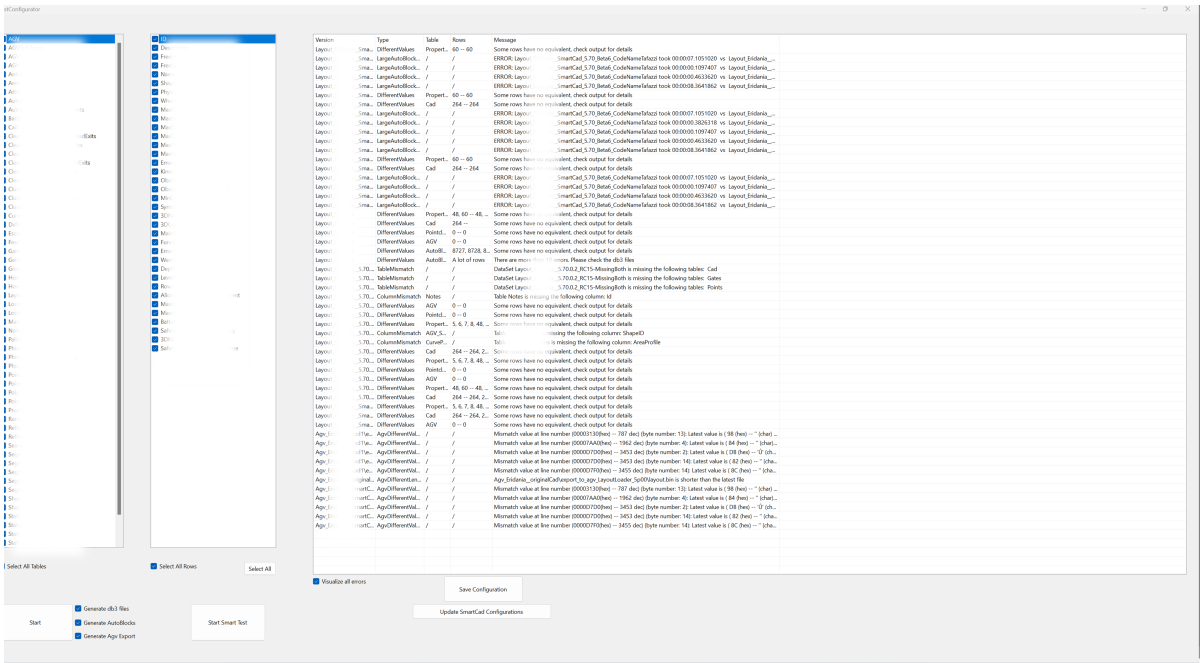

Figure 7.3: GUI

As it's depicted in the above image there are two columns on the right that represent the

tables with their relative fields. As the GUI is loaded the tool reads the model file and updates the first column on the left with the tables' names whilst the second is dynamically updated as the first column's selected item is changed. As we have already said before, we wanted our applicative to be fully customizable so every table and every field can be easily deselected within a mouse click. This will allow specific functionality testing and it will end up saving time. Three buttons are available to easily select all tables, fields in a table or both.
Near the Start SmartCad Test button there are 3 check-boxes to choose if those three features have to be tested during the current test. The Save Configuration button allows to save that configuration in the JSON configuration files, saving all the tables that will be tested and all the other parameters.
Clicking on the Update SmartCad Configurations opens up another window that shows where the user saved all their layouts, Smart and SmartCad folders, etc. The interface is quite easy to understand and if a path wants to be updated it just require the user to click on the button Change and select the path from a friendly File Browser Dialog window.

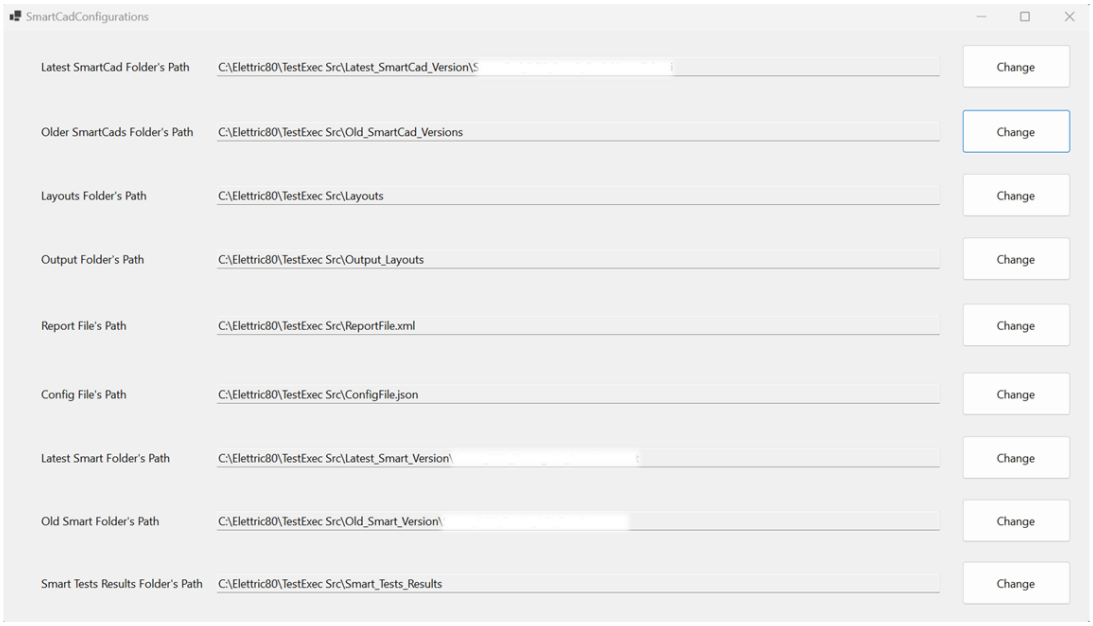

Figure 7.4: GUI Configuration Window

Taking a step backwards to figure 7.3 we can see that there is a ListView which is used to display in a summarized way all the errors that may arise. It will display the version of the Smart or SmartCad that generated the error, the type of error, which table and/or

which rows are at fault and what is the error message that is returned. Underneath there is a check-box to decide if all mistakes have to be visualized, this is because errors are divided into relevant and less relevant depending on what their type is.
Finally there are the two buttons that will either start the Smart or the SmartCad test.

7.6. CLI

Once a configuration file has been modified to suit the user's necessities and the usage of a graphical interface is considered superfluous we can move over our Command-Line Interface. It is designed to launch both the tests with the option of using different configuration files using one single and easy command. If we want to launch the Smart test we can just add the parameter `Sim`, this will utilize the default configuration that we saved through our graphical interface. If we want to utilize another configuration file but we don't want to re-open the GUI, we can add a second parameter to the arguments. For example our parameters will become `Sim ConfigFileName`, where ConfigFileName is the name of a configuration file saved in a default directory under C. Trying to launch the SmartCad test is approximately the same as we have to change just the Sim keyword into `Cad`.
Some examples would be:

```
> ABCC.exe cad
> ABCC.exe sim
> ABCC.exe sim Config2.json
```

From the CLI it's easy to understand the tool's workflow. Thanks to a constant update on what the tool is evaluating at the moment we always know which layout is in process and which SmartCad version is used for it. We also know in which step we are in, e.g. if we are loading a layout, comparing DataTables or else. Thanks to a progression bar we can also figure out how many steps and how much time we have left before the test for that specific layout is completed.

Figure 7.5: CLI Workflow

Every type of error will be reported to output in the most human-interpretable way, facilitating the user in the comprehension of the error. In the image below there is an example of some errors that may occur and how they are visualized.

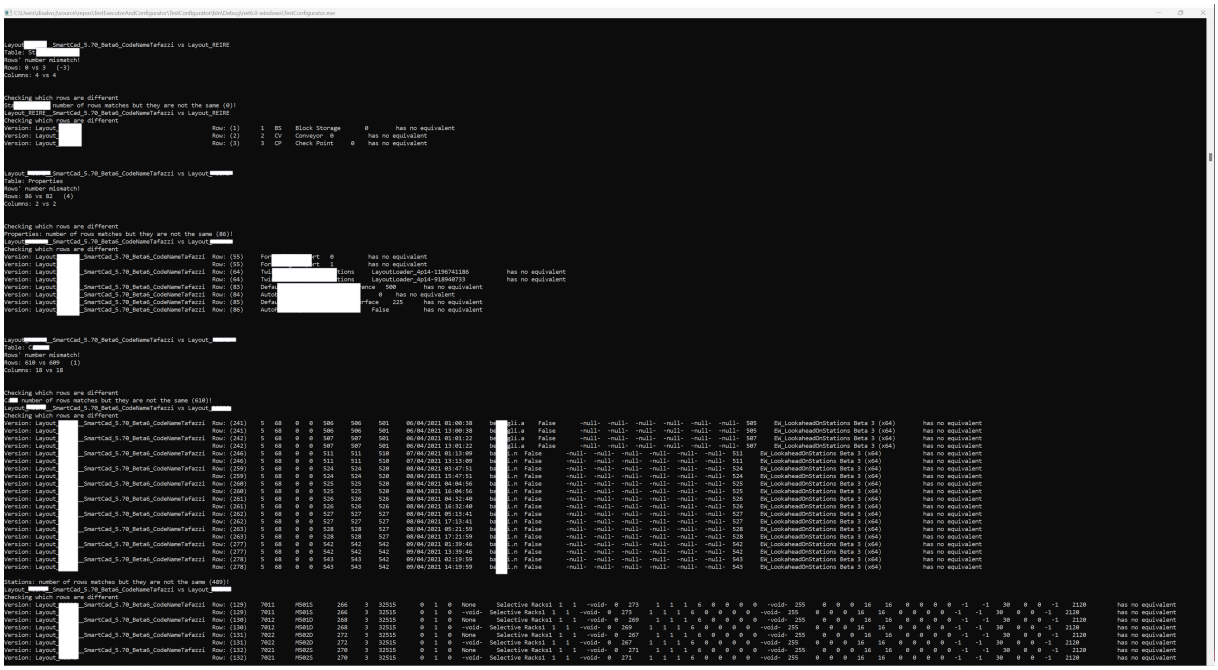

Figure 7.6: CLI Error Visualization

7.7. Report

Our report file will be written in eXtensible Markup Language (XML) to be easily interpreted by a human, as you can easily move through the `XElements` by minimizing and maximizing within one click. It is also useful because in future the same file can be loaded in a ReportViewer that will extract info from the file and execute statistic on that data. It's structured to have two main tags: `Summary` and `Details`. Summary keeps track of every test case analyzed and stores a positive response if the confrontation with that file/simulation gave no errors back, otherwise it stores a negative response. Details keeps track just of the test cases that returned one or more errors, indicating what type of error it was and some info to better understand it. The image below represents a report file:

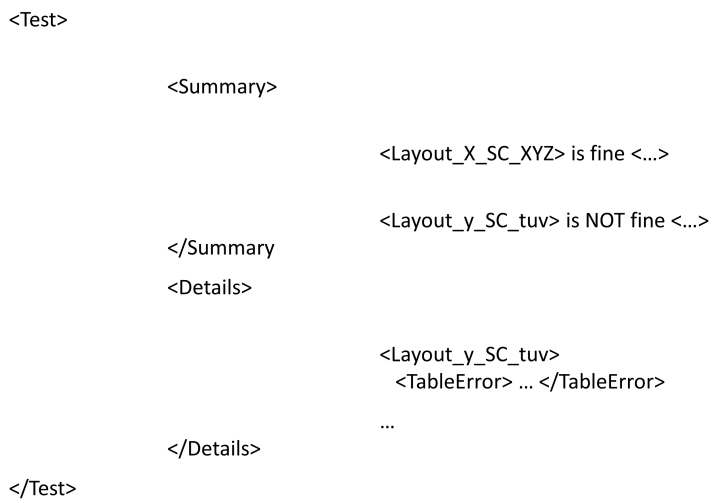

Figure 7.7: Report Structure

8 | Evaluation

In this chapter I will try to evaluate the overall results, proceeding for each of our requirements.

8.1. Test Case Generation

In accordance with the specifications outlined in section 2.1 of the document, the tool is designed to facilitate a seamless interaction with two fundamental components, namely Smart and SmartCad. These components collectively contribute to the tool's capability to generate an extensive array of test cases, a process meticulously carried out to ascertain the proper functioning of specific as well as critical functionalities that have been explicitly solicited by the user.

8.2. Immediate Response

This requirement [2.2] is achieved by exploiting all the possible parallelism between tasks and by using the most efficient comparison algorithms. As it's shown in Chapter 7 whether it'd be through graphical or command-line interface or through an Xml file a positive or negative result will be returned, underlining the errors that may have occurred.

8.3. Representation of Data

We managed to fulfill this requirement [2.3] through an adaptation of our code to manage all the demanded types of data. We utilized JSON files to stash the configurations of our program and to save information relevant to the test cases. We used the Xml format to represent the outcome of the test cases that we observed. Db, txt and bin files have all been used both in reading and in writing.

8.4. SmartCad Tester

Satisfying requirement 2.4 is accomplished by running the SmartCad test as illustrated in section 6.6. Once a layout or an AGV export file has been generated then our only goal is to assure that it's equal or as equal as possible to the expected output created by one of the stable version of SmartCad.

8.5. Smart Tester

Contrary to the previous requirement this one [2.5] is accomplished not during the SmartCad test but during the Smart test. We monitor the software's behaviour to detect if any unexpected behaviour arises, noting minor errors and aborting if major errors show up.

8.6. Backward Compatibility Rules

There are some hard-coded backward compatibility rules as some time thresholds that shall not be passed. Other compatibility rules are decided by the users bwfore they start the application, for example they can exclude the confront on the field "Colour" in table "AGV" because they already know it would change therefore preventing time consumption and a failure on the final report. This way we managed to fulfill requirement 2.6

8.7. Reporting And Analysis

As required in section 2.7 we managed to create comprehensive and easily understandable reports highlighting compatibility risks and potential issues. This has been achieved through our Command-Line Interface [7.6], through our graphical interface using a ListView [7.3] and last but not least through our report [7.7].

8.8. Scalability And Performance

Requirement 2.8 is reached by exploiting parallelism, spawning more tasks as the workload increases.

8.9. User-Friendly Interface

Thanks to our GUI [7.5] also this requirement is fulfilled. We have seen how easy it is to change our configurations and how intuitive it is to view and understand the results.

Minimal training is required and this document along with a summary are more than enough to understand how to use the tool.

8.10. Command-Line Usage

What we have seen in section 7.6 is more than enough to decree the requirement 2.10 as checked. It's intuitive and easy to launch with almost no parameters to add, avoiding confusion.

9 | Conclusions and future developments

9.1. Summary

The idea of project ABCC was to create a complete tool able to backward compatibility in an automated way. We have seen that it's possible to provide a solution for our purpose without paying for commercial software, but also create a consistent software that can handle ad hoc input and test through a collection of critical functionalities that the two company software expose. Being SQLite and NLog the only open-source software that we relied upon and being NuGet packages allows an easy management of the two. The major advantage of having developed our own software for the front end and back end parts of the central tool, is that we can continually add new features as needed and expand it as we want. Moreover, since we used C# as programming language all are code is object-oriented. This makes our code is fragmented in more pieces which allow easy modifications. New feature can be tested just by adding more classes and just marginally modifying the old code.

9.2. Future Work

I'm glad to say that ABCC has already been used in the office to test some release candidate versions for SmartCad. They underlined an issue with the introduction of multitasking in a specific part of the code that reduced or multiplied the available data in the layout. The project is still growing and in future its architecture may risk to be modified, to better adapt to the new functionalities that may be released or to the available functionalities that already exist but are not yet being tested. If we want to better exploit parallelism we could have to leave behind SQLite and the DataSet structure in favor a solution that better fit concurrent writing.
Future work may comprehend the introduction of new testing features and better performance supervision. New API endpoints will be added to interact with the system,

to better oversee the dynamic simulation of the software. One last aspect is to facilitate seamless integration into the software development workflow, the tool should then be compatible with continuous integration/continuous deployment (CI/CD) pipelines. It should support automation of compatibility testing during the build and release processes. This tool is design to reduce to the minimum the effort for the developers.

Bibliography

[1] Dataset, . URL https://learn.microsoft.com/en-us/dotnet/api/system.data.dataset?view=net-7.0.

[2] Datatable, . URL https://learn.microsoft.com/en-us/dotnet/api/system.data.datatable?view=net-7.0.

[3] Hashset. URL https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-7.0.

[4] Nuget. URL https://www.nuget.org/.

[5] Task. URL https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-7.0.

[6] Winmerge project repository on github. URL https://www.sqlite.org/sqldiff.html.

[7] Sqldiff, 2000. URL https://www.sqlite.org/sqldiff.html.

[8] P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

[9] M. Khan et al. Different approaches to black box testing technique for finding errors. *International Journal of Software Engineering & Applications (IJSEA)*, 2(4), 2011.

[10] M. E. Khan and F. Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, 3(6), 2012.

[11] M. E. Khan et al. Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications*, 5(3):1–14, 2011.

[12] R. Poston and M. Sexton. Evaluating and selecting testing tools. *IEEE Software*, 9 (3):33–42, 1992. doi: 10.1109/52.136165.

[13] M. Shi. Software functional testing from the perspective of business practice. *Computer and information science*, 3(4):49, 2010.