

Politecnico di Torino

Politecnico di Torino

Dipartimento di Elettronica e Telecomunicazioni (DET)

Corso di Laurea Magistrale in Ingegneria Elettronica
(Electronic Engineering)

TESI DI LAUREA MAGISTRALE

Automated ATPG Run Monitoring, Debug, and Analytics Tools for Large Scale ATPG Runs For Complex Designs

Relatori:

Prof. Guido MASERA

Eng. Anthony WALSH

Candidata:

Mariagiusy BALDASSARRA

Sessione di Laurea Settembre 2026

Abstract

Modern semiconductor manufacturing relies on Design-for-Test (DFT) methodologies to ensure integrated circuits meet quality standards. A critical step is Automatic Test Pattern Generation (ATPG), which creates test sequences to detect manufacturing defects. In industrial settings, ATPG runs can take hours or days, generating extensive log files with thousands of lines of operational data, timing information, and quality metrics. Manually analyzing these files is impractical and error-prone, especially when multiple jobs run concurrently across different designs.

This thesis presents the ATPG Run Analyzer, a Python-based tool that automates the extraction, analysis, and reporting of critical information from Siemens Tessent ATPG log files. The tool operates through two primary modes: Profiler mode for completed runs and Monitor mode for active runs.

Profiler mode performs deep analysis of completed log files, parsing timestamps and phase boundaries to calculate elapsed times and extract per-iteration coverage metrics. It generates multi-sheet Excel reports with embedded visualizations and HTML email summaries. When provided with a reference baseline, it performs comparative analysis, highlighting deviations in timing, coverage, and pattern count through color-coded outputs.

Monitor mode provides real-time oversight of ongoing ATPG jobs through periodic log file polling. It detects three classes of anomalies: stuck runs where the log file stops growing, slow tasks exceeding expected duration thresholds, and low coverage when quality targets are not met. The Monitor operates with or without reference baselines, using relative comparison against historical data when available or absolute thresholds for new designs.

Both modes share a common reference management system that maintains baseline datasets from successful runs, capturing task durations, coverage evolution, and comprehensive metrics. This enables meaningful comparison across runs and facilitates early detection of performance regressions.

The tool incorporates Test-Driven Development practices with a comprehensive unit test suite of 69 tests that executes automatically at startup, verifying parsing logic, time calculations, and metric extraction. This ensures reliability in industrial environments where incorrect analysis could lead to wasted computational resources or missed defects. By automating routine analysis tasks and providing proactive monitoring, the tool contributes to a more efficient and reliable semiconductor testing process.

Contents

1 Introduction **4**

1.1 Objectives of this work 4

1.2 Thesis structure 5

2 Design For Test **6**

2.1 Reasons behind DFT 6

2.2 Fault models 7

2.3 Automatic Test Pattern Generation 8

2.4 Automatic Test Equipment 9

2.5 Scan Design 9

2.6 Boundary Scan 10

2.7 Built-In Self-Test 11

2.8 Memory BIST 12

2.9 DFT in the design flow 13

3 Siemens Tessent Platform **14**

3.1 Key Tessent commands 15

3.2 EDT and scan compression 19

3.3 Test patterns generation with Siemens Tessent 19

3.4 Analysis of *report_statistics* 22

3.5 Tessent ATPG log files 23

3.5.1 Structure and format 23

3.5.2 Per-iteration metrics 24

4 Software Quality and Unit Testing **25**

4.1 Test-Driven Development Methodology 26

4.2 Python unittest 27

4.3 Structure of the test suite 28

4.4 Testing pure functions 28

4.5 Testing file I/O 30

4.6 Regression testing with reference files 31

4.7 Using unittest.mock 33

4.8 Why this testing approach is useful . . . 36

5 ATPG Run Analyzer — Overview and Architecture 37

5.1 System architecture . . . 37
5.1.1 High-level design . . . 38
5.1.2 Technology stack . . . 38
5.1.3 Module structure . . . 39
5.2 Entry point and startup sequence . . . 40
5.3 The main menu . . . 40
5.4 Profiler mode . . . 41
5.4.1 Purpose . . . 41
5.4.2 User inputs . . . 41
5.4.3 How log parsing works . . . 42
5.4.4 Handling time calculations across calendar boundaries . . . 42
5.4.5 Extracting coverage and pattern data . . . 43
5.4.6 Analysis process . . . 44
5.4.7 Output structure . . . 44
5.4.8 Incomplete run handling . . . 45
5.5 Monitor mode . . . 45
5.5.1 Detecting stuck runs . . . 47
5.5.2 Detecting slow tasks . . . 48
5.5.3 Detecting low coverage . . . 49
5.5.4 Email alert system . . . 49
5.6 Updating references . . . 50
5.7 Configuration file . . . 52
5.8 Summary . . . 52

6 Results and Tool Validation 53

6.1 Profiler mode results . . . 53
6.1.1 Analysis of a complete ATPG run . . . 53
6.1.2 Comparative analysis with reference baseline . . . 56
6.1.3 Incomplete run handling . . . 59
6.2 Monitor mode results . . . 60
6.2.1 Monitoring with reference baseline . . . 60
6.2.2 Monitoring without reference baseline . . . 62
6.3 Time savings and efficiency gains . . . 63

7 Conclusions 64

Bibliography 69

Chapter 1

Introduction

The semiconductor industry has seen extraordinary growth in the complexity of integrated circuits over the past few decades. Modern chips integrate billions of transistors [1], embedding complex logic, large memory arrays, and a wide variety of functional blocks into a single device. Ensuring that all of these components work correctly after manufacturing is not a trivial task, and it is precisely here that Design for Test (DFT) plays a fundamental role. Within the DFT flow, Automatic Test Pattern Generation (ATPG) is one of the most computationally demanding steps. It involves generating and validating the test patterns used to detect manufacturing defects, and in industrial settings it can take many hours or even days to complete. During this process, the ATPG tool continuously logs its activity into log files: text files that record every operation performed, from design loading and rule checking to pattern generation, fault simulation, and final coverage results. These log files contain a wealth of useful information. In principle, they tell the engineer exactly what happened during a run, how long each phase took, and what results were achieved. In practice though, a single log file can easily span tens of thousands of lines, mixing critical metrics with low-level operational messages. Reading through them manually is tedious, slow, and error prone. When multiple runs are being executed simultaneously across different designs keeping track of everything becomes simply unmanageable. There is also another important limitation worth highlighting. When a run is still in progress, there is no straightforward way for an engineer to know if something is going wrong. A task might take far longer than expected, or the run might have stalled entirely. Without actively monitoring the log file in real time, these issues often go unnoticed until the job finishes or crashes, wasting valuable computing time and resources.

1.1 Objectives of this work

This thesis presents the ATPG Run Analyser, a tool developed during an internship in the Infra-DFT team at Qualcomm Ireland. The goal of the tool is to automate the analysis and monitoring of Siemens Tessent ATPG log files, addressing the limitations described above. The tool is designed to handle both completed and actively running ATPG jobs, and its main features are:

- **Automated log parsing**: Extracting and structuring the key metrics from a log file automatically, without any manual inspection.
- **Detailed Excel reports**: Presenting the parsed data in a clear, multi-sheet Excel report that allows engineers to quickly evaluate run performance and identify bottlenecks.
- **Email summary reports**: Sending a concise summary of the results via email upon run completion, so engineers can stay informed without having to open any files.
- **Real-time alerts**: Monitoring runs that are still in progress and sending email alerts if something unexpected is detected, such as a task taking unusually long or the run appearing to stall.
- **Reference comparison**: Comparing a run against a known good (golden) reference, highlighting any phases that deviate significantly from the expected baseline and helping engineers quickly spot regressions.

Together, these features aim to reduce the time engineers spend on manual log inspection, improve visibility into both ongoing and completed ATPG jobs, and ultimately contribute to a more efficient and reliable DFT workflow.

1.2 Thesis structure

The rest of this thesis is organized as follows:

- **Chapter 2** provides theoretical background on Design for Test, covering the main concepts and techniques that form the foundation of modern chip testing.
- **Chapter 3** moves into the industrial context, introducing the Siemens Tessent platform and its toolchain, describing how ATPG runs are managed in practice, and explaining how this directly motivates the need for the tool developed in this work.
- **Chapter 4** discusses the role of unit testing in software development, with a focus on the Python testing framework used throughout the project, highlighting how a structured testing approach contributes to building reliable and maintainable tools.
- **Chapter 5** presents the implementation of the ATPG Run Analyser tool, detailing the architecture, design decisions, and key components that enable automated log parsing, monitoring, and reporting.
- **Chapter 6** presents the results obtained from deploying the tool in the industrial environment.
- **Chapter 7** concludes the thesis with a summary of the work done and outlines potential future developments.

Chapter 2

Design For Test

Design-for-Test (DFT) is a methodology that incorporates testing considerations directly into the early stages of the hardware design process. By adding specific testing features to a chip before it is manufactured, DFT ensures that the final product can be evaluated quickly and accurately, significantly reducing both testing costs and time-to-market.

A key concept in this process is testability, which measures how easily a manufactured chip's quality can be verified. For a design to be highly testable, it must be both controllable and observable. This means test engineers must be able to easily manipulate the internal states of the chip (controllability) and monitor the resulting behaviour (observability) to uncover any manufacturing defects. Ultimately, DFT makes it much easier to identify and diagnose hardware issues early in the production cycle.

There are two primary approaches to DFT:

- **Ad hoc**: This approach relies on applying general good design practices to improve testability. Examples include removing unnecessary logic, separating clock signals, or adding simple observation points without fundamentally changing the chip's architecture.
- **Structured**: This approach involves systematic, automated modifications to the chip's design to support comprehensive testing. Key techniques include modifying the internal circuitry to make it easier to test (Scan Design), building autonomous testing capabilities directly into the device (Built-In Self-Test, or BIST), or adding dedicated logic to test connections at the board level (Boundary Scan).

2.1 Reasons behind DFT

The need for DFT is fundamentally driven by the imperfections of the semiconductor manufacturing process. Even in state-of-the-art fabrication facilities, microscopic anomalies, such as particles, chemical impurities, or lithographic irregularities, can occur during production and introduce physical defects into the silicon. These defects directly impact yield, defined as the percentage of manufactured chips that function correctly and meet their specifications. Since

defective chips must be identified and discarded before reaching the customer, testing is an essential and unavoidable step in the production cycle. The goal is therefore not only to detect these defects, but to do so as efficiently and reliably as possible which is precisely what DFT methodologies are designed to enable.

2.2 Fault models

Since manufacturing processes are imperfect, they can introduce physical defects that prevent a circuit from functioning properly. To abstractly represent these physical defects within the logical model of a design, engineers use fault models. A fault model defines what can go wrong, how this incorrect behavior manifests, and what constitutes a successful or failed test.

Once the ”fault universe” (the total number of possible faults in a design) is defined, the effectiveness of a testing strategy can be measured numerically. A high fault coverage indicates a robust testing process; however, this assessment is only possible if physical defects are accurately translated into logical faults. In essence, the complex problem of physical defect detection is transformed into the more manageable problem of logical fault detection.

A fault site is a specific location in the circuit where a fault is known or assumed to exist. Identifying these sites allows engineers to emulate specific errors within the circuit’s design to see how the system reacts.

While there are multiple classes of fault models, the two most common are:

- Stuck-at faults
- Transition faults

The stuck-at fault model assumes that a manufacturing defect causes a specific signal line to be permanently stuck at a constant logic value, regardless of the inputs applied. This includes:

- **Stuck-at-0 (SA0)**: The signal is permanently tied to logic 0
- **Stuck-at-1 (SA1)**: The signal is permanently tied to logic 1

While stuck-at faults model static errors, the transition fault model addresses timing or delay defects. In this model, the signal can change its value, but it fails to make the transition within the required time limit (typically a single clock cycle). This is categorized into:

- **Slow-to-rise**: The signal takes too long to transition from a logic 0 to a logic 1
- **Slow-to-fall**: The signal takes too long to transition from a logic 1 to a logic 0

Once these fault models are defined and the fault sites are identified, the next challenge is creating the specific input signals required to expose them. To test for a stuck-at or transition fault, a specific sequence of inputs must be applied to force the faulty node to a certain value and observe the result. Because modern integrated circuits contain millions or billions of gates, generating these test sequences manually is impossible, hence a way to automatically find these patterns is required.

2.3 Automatic Test Pattern Generation

The main goal of Automatic Test Pattern Generation (ATPG) is to create an optimized set of test patterns that achieves a target test coverage, defined as the percentage of all testable faults that the pattern set successfully detects. These test patterns, also referred to as test vectors, are specific sequences of logical 1s and 0s applied to the chip's input pins during the manufacturing test to verify its correct operation.

To achieve high test coverage efficiently, the ATPG process fundamentally relies on two steps:

- **Pattern Generation**: The algorithmic process of calculating the specific input stimulus required to activate a targeted fault and propagate its effect to a point where it can be observed.
- **Fault Simulation**: The evaluation of these generated patterns against the broader fault universe. This step determines exactly which additional faults are incidentally detected by the generated pattern, which prevents redundant calculations and keeps the final pattern set as compact as possible to minimize expensive testing time.

When generating new patterns internally, ATPG typically relies on two complementary methods:

- **Random Pattern Test Generation**: This method generates a sequence of pseudo-random patterns and relies on fault simulation to identify which ones happen to detect faults. Only the patterns that successfully expose faults are retained and stored in the test set. Utilizing a small number of random patterns as an initial step is a highly efficient way to clear out a large volume of easily detectable faults, thereby significantly improving overall ATPG performance and saving computational time.
- **Deterministic Pattern Test Generation**: To address the remaining faults that random patterns miss, this method calculates a specific test pattern intended to target a given fault. The deterministic procedure involves systematically picking an undetected fault from the fault list, mathematically deriving a pattern designed to detect it, and then fault-simulating that newly created pattern to verify its effectiveness before adding it to the final test set.

In addition to generating their own vectors, modern ATPG tools can fault-simulate patterns provided from an external set. The tool evaluates these external patterns, identifies which ones successfully detect faults, and retains only those effective patterns for the final test set.

By combining these techniques, the ATPG process ultimately produces a comprehensive and compact set of test data. This data is then translated into specific formats required by Automatic Test Equipment (ATE), enabling the external tester to accurately apply the patterns to the physical chip. However, in modern integrated circuits with millions of gates, relying solely

on a chip's standard external input and output pins to apply these deterministic and random patterns is highly impractical. Accessing deep internal logic requires a structural modification to the circuit.

2.4 Automatic Test Equipment

Once the ATPG process is complete, the generated test patterns must be physically applied to the chip. This is the role of Automatic Test Equipment (ATE), a specialized external testing machine used in manufacturing facilities to interface directly with the chip under test. The ATE loads the test patterns onto the chip's input pins, measures the resulting output responses, and compares them against the expected values to determine whether the device passes or fails. While ATE is an indispensable part of the manufacturing test flow, it comes with significant costs: ATE machines are expensive to procure and operate, and test time on an ATE directly translates to cost per chip. This economic pressure is one of the key motivations behind techniques such as scan compression and Built-In Self-Test (BIST), which aim to reduce the time and reliance on external testers.

2.5 Scan Design

Scan Design is a structured DFT methodology that addresses this challenge directly. It works in two key steps:

1. **Replacement**: All memory elements in the design are replaced with their scannable equivalents, like the one in Figure 2.1 , commonly known as scan cells or scan flip-flops. These are enhanced versions of standard flip-flops that, in addition to their normal functional operation, include a dedicated shift mode that allows data to be serially loaded and read out.

2. **Stitching**: The scan cells are then connected into one or more scan chains, essentially long shift registers that run through the chip. This gives the tester a direct path to control and observe the value stored in every memory element in the design.

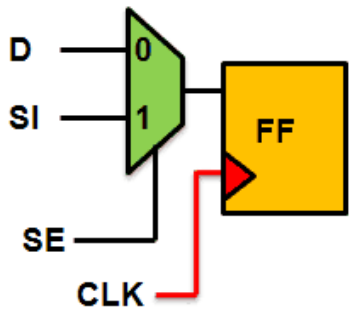


Figure 2.1: Scan Flip-Flop with additional signal: SE (scan enable), SI (scan in) and SO (scan out) [2]

By providing full controllability and observability over all internal storage elements, Scan Design effectively transforms the testing problem. What was once a complex sequential circuit test becoming as straightforward as testing a combinational circuit, greatly simplifying both test pattern generation and fault simulation for ATPG tools.

While Scan Design dramatically improves the testability of a chip, it introduces a practical challenge of its own. As designs grow in complexity and the number of scan cells increases, scan chains become extremely long. This results in a very large number of test patterns that must be shifted in and out of the chip via the ATE, leading to excessive test times and consequently higher manufacturing costs.

To address this, modern DFT flows incorporate scan compression techniques. The core idea is to insert dedicated on-chip compression and decompression logic between the ATE and the chip's internal scan chains, allowing a small number of external scan channels to control a much larger number of internal scan chains simultaneously. From the perspective of the ATE, the chip appears to have only a few short scan chains, while internally a far greater number of chains are being driven and observed in parallel. This significantly reduces the volume of data the tester needs to handle, cutting both test time and ATE memory requirements without sacrificing fault coverage.

2.6 Boundary Scan

Boundary Scan is a testing method designed to verify the integrity of interconnections, the physical wire traces that link chips together on a Printed Circuit Board (PCB), as well as the sub-blocks within an individual integrated circuit. Beyond manufacturing test, it is also widely adopted as a powerful debugging tool during board bring-up and system validation.

The fundamental idea behind Boundary Scan is to eliminate the need for physical test probes. Instead, a dedicated Boundary Scan Cell is added to each input and output pin of a chip. These cells sit between the chip's core logic and its external pins and can be individually programmed to either drive a known value onto a pin or capture the value currently present on it. This gives engineers full control and observability over every pin on the device without

ever needing to physically touch them.

Testing an interconnection between two chips is therefore straightforward: a value is driven onto the source pin via its boundary scan cell, and the cell at the destination pin reads back the result. If the expected value is not received due to an open circuit, a short to another signal, or any other physical defect, a fault is immediately reported.

All boundary scan cells within a chip are linked together into a scan chain, which is controlled through the JTAG interface, an industry-standard protocol formally defined by the IEEE 1149.1 standard. JTAG provides a simple, low-overhead serial communication interface that allows a tester to access and control the chip's internal test logic without requiring direct access to the system's address or data buses.

This serial interface connects to an on-chip block called the Test Access Port (TAP), which acts as the gateway to all boundary scan operations. The TAP is governed by a dedicated state machine, the TAP Controller, which manages access to a set of internal test registers that expose the logic states and capabilities of different parts of the chip.

2.7 Built-In Self-Test

While Scan Design greatly improves the testability of a chip's internal logic, it still relies on an external tester to supply and evaluate the test patterns. As chips grow increasingly complex, this dependency becomes a bottleneck, external testers are expensive, and the volume of patterns required can make testing very time-consuming. A natural evolution of DFT is therefore to move the testing capability inside the chip itself, which is the core idea behind Built-In Self-Test (BIST): embedding dedicated hardware directly onto the chip that can autonomously generate test patterns, apply them to the circuit under test, and evaluate the results, all without relying on external equipment.

A typical BIST architecture consists of three main components:

- **Pattern Generator**: A hardware block responsible for producing test stimuli to be applied to the circuit. This is commonly implemented as a Pseudo-Random Pattern Generator (PRPG), which efficiently generates many pseudo-random test vectors using a small amount of hardware.
- **Circuit Under Test (CUT)**: The portion of the chip being tested, which receives the patterns produced by the generator.
- **Output Response Analyzer (ORA)**: A hardware block that compresses and evaluates the output responses of the circuit under test. A commonly used implementation is a Multiple Input Signature Register (MISR), which compacts the entire output response into a compact signature that can then be compared against a known-good reference value to determine whether the circuit has passed or failed.

Based on the type of circuit being tested, BIST is broadly categorized into two types:

- **Logic BIST (LBIST)**: Targets the testing of combinational and sequential logic within the chip. It works in close conjunction with the chip's scan chains, using the PRPG to load patterns into the scan cells and the MISR to collect and compress the output responses.
- **Memory BIST (MBIST)**: Targets the testing of embedded memories (such as SRAMs and ROMs), which are particularly sensitive to manufacturing defects. MBIST applies specialized memory test algorithms to verify the correct read and write behavior of every memory cell.

The key advantage of BIST is its flexibility: it can be triggered not only during manufacturing tests, but also during system bring-up or even in the field, enabling continuous health monitoring throughout the lifetime of the device.

2.8 Memory BIST

Unlike Logic BIST, which relies on pseudo-random patterns, MBIST employs structured deterministic algorithms specifically designed to stress and expose the failure modes of memory arrays. The most widely used class of these are March algorithms, which apply a defined sequence of read and write operations to every memory cell in a specific order (e.g., incrementing or decrementing address sequences). Well-known examples include March C- and March B, each targeting different classes of memory faults such as stuck-at faults, transition faults, and coupling faults between adjacent cells.

At the heart of the MBIST system is the MBIST Controller, a dedicated hardware block inserted into the chip whose primary role is to orchestrate the entire self-test process for all embedded memories under its supervision. Specifically, the MBIST controller is responsible for:

- **Algorithm execution**: Loading and executing the selected memory test algorithm, driving the correct sequence of addresses and read/write operations to the memory under test.
- **Address and data generation**: Generating the precise row and column addresses, writing data patterns, and expected read-back data required by the test algorithm.
- **Response checking**: Comparing the actual data read back from memory against the expected values, flagging any mismatches as failures.
- **Result reporting**: Collecting pass/fail status and, in more advanced implementations, logging detailed failure information (such as failing address and data) to support diagnosis and repair.

The MBIST controller communicates with the memories through a dedicated test interface and is typically accessed and controlled via a standard JTAG (IEEE 1149.1) interface or a similar

test access port. This means the MBIST test can be initiated not only during manufacturing test on an ATE, but also during silicon bring-up, system verification, or even in the field enabling continuous health monitoring throughout the lifetime of the device.

In chips containing many embedded memories, it is common to deploy multiple MBIST controllers, each responsible for a dedicated subset of memories, allowing tests to be run in parallel to significantly reduce overall test time.

2.9 DFT in the design flow

It is worth noting that DFT is a structured activity integrated into the chip design flow at a well-defined stage. DFT insertion typically takes place after logic synthesis, once the design has been translated into a gate-level netlist, and before physical implementation (place-and-route). At this stage, scan cells replace standard flip-flops, BIST logic is inserted, and boundary scan infrastructure is added. Increasingly, DFT planning also begins much earlier in the design cycle, a practice known as DFT-aware design, to ensure that architectural decisions do not inadvertently compromise testability and to avoid costly design re-spins later in the process.

Chapter 3

Siemens Tessent Platform

The DFT techniques described in the previous chapter are, in an industrial setting, implemented and managed through specialized Electronic Design Automation (EDA) tools. Among the most widely adopted platforms for this purpose is Tessent, developed by Siemens EDA (formerly Mentor Graphics). Tessent is a comprehensive DFT toolchain that covers the full spectrum of test insertion, pattern generation, memory testing, and diagnosis, and it is the platform used as the basis for the work presented in this thesis.

At the core of the Tessent environment is Tessent Shell, a unified command-line interface through which all Tessent tools are invoked and controlled. Rather than operating as a collection of standalone applications, the different Tessent technologies run as modules within this shared shell environment, each activated by setting a specific context. This design gives the platform a consistent user experience and makes it straightforward to integrate different DFT tasks within the same session. The shared design and test data is stored in a directory structure called the Tessent Shell Database (TSDB), which allows different tools in the flow to exchange information seamlessly without the need for manual data conversion steps.

The main tools available within the Tessent platform are the following.

- **Tessent Scan** handles the automatic insertion of scan circuitry into the design. It identifies all sequential elements that are candidates for scan replacement, converts them into scan cells, and stitches them into scan chains. It also performs Design Rule Checking (DRC) to verify that the scan architecture is correctly configured before any patterns are generated. The tool produces a Test Configuration Data (TCD) file that carries all the necessary scan configuration information for the downstream ATPG tools to use.

- **Tessent ATPG (TestKompress)** handles the generation and simulation of test patterns. It reads the design netlist, the cell library, and the scan configuration produced by Tessent Scan, and runs the full ATPG flow, including random pattern generation, deterministic pattern generation, and fault simulation, to produce a compact and high-coverage set of test patterns. Tessent TestKompress also integrates scan compression logic directly

into this flow through its EDT technology, which is discussed in more detail later in this chapter.

- **Tessent MemoryBIST** handles the insertion of Memory BIST infrastructure for embedded memories. It generates and inserts the MBIST controllers responsible for running structured test algorithms on SRAMs and ROMs present in the design.

- **Tessent Boundary Scan** handles the insertion of boundary scan cells and JTAG infrastructure in compliance with the IEEE 1149.1 standard, enabling board-level interconnect testing as described in the previous chapter.

Beyond test insertion and pattern generation, the Tessent platform also includes tools for post-silicon activities. Tessent Diagnosis analyses failure data collected by the ATE to identify the likely physical locations of defects on a failing chip. Tessent SiliconInsight provides an interactive environment for debugging BIST-tested memories and logic directly on silicon. Tessent YieldInsight supports statistical yield analysis based on diagnosis results, helping teams identify systematic failure patterns across large production volumes.

All these tools share the same shell environment, the same command syntax, and the same underlying data model. As a result, the log files they produce during a run follow a consistent and predictable structure. This is a key observation for the work presented in this thesis, as it forms the foundation on which the ATPG Run Analyser is built.

3.1 Key Tessent commands

Tessent Shell uses a Tcl-based scripting interface, where commands are organized around the current system mode (setup or analysis) and the active context. Understanding the most used commands is essential to following how a typical ATPG run is structured, and consequently how its log file is generated and interpreted.

The overall flow begins by setting the working context, then loading the design, performing rule checks, and finally entering analysis mode to run pattern generation. The most important commands in this flow are described below.

- **set_context** sets the operating context for the current Tessent session. This determines which tool and which test mode are active. For example:

  ```
  set_context patterns -scan
  ```

  This activates the ATPG context for scan-based pattern generation. Different contexts are used for scan insertion, MBIST, boundary scan, and so on.

- **read_verilog** (or read_flat_model) loads the design netlist into the tool. The flat model is a pre-compiled, optimized representation of the design that is faster to load in large industrial flows:

```
read_verilog my_design.v
```

- **set_system_mode** switches the tool between its two operating phases. The setup mode is used for loading data and configuring the session, while the analysis mode is where rule checks are finalized and the actual ATPG computation takes place:

```
set_system_mode analysis
```

- **set_fault_type** specifies which fault model to use for the ATPG run. By default, the tool targets stuck-at faults, but this command allows switching to transition faults, path delay, or other models:

```
set_fault_type transition
```

- **add_faults** populates the internal fault list with all faults of the currently selected type. This is typically one of the first commands issued after entering analysis mode:

```
add_faults -all
```

  It is also possible to selectively add faults on specific instances or pin pathnames, rather than the entire design, which is useful when targeting specific areas for coverage improvement.

- **read_faults** loads a fault list from an external file into the tool's internal fault list. This is commonly used to carry over fault status information from a previous run, so that already-detected faults are not targeted again unnecessarily. The -retain option preserves the fault classification codes from the file:

```
read_faults previous_run.faults.gz -retain
```

- **write_faults** saves the current internal fault list to an external file. This is used at the end of a run to archive the fault status, share results between runs, or feed fault information into downstream tools such as Tessent Diagnosis:

```
write_faults output.faults.gz -replace
```

- **report_faults** displays information about the faults currently in the internal fault list, filtered by their classification. For example, it can be used to inspect which faults are still undetected after pattern generation:

  ```
  report_faults -class UC
  ```

- **create_patterns** is the core ATPG command. It triggers the full pattern generation flow, combining random and deterministic generation with integrated fault simulation, until the target coverage is reached or no further progress can be made. In its simplest form it is invoked with no arguments:

  ```
  create_patterns
  ```

  However, it supports several important options that allow engineers to fine-tune its behaviour:

  – **-auto**: instructs the tool to automatically manage the entire pattern generation process, including all internal ATPG phases, without requiring manual intervention. It is typically used when a fully automated flow is desired. It is worth noting that using this option can sometimes lead to higher pattern counts compared to more carefully tuned manual flows, as the tool makes conservative default choices internally.

    ```
    create_patterns -auto
    ```

  – **-compaction_effort**: controls how hard the tool tries to reduce the final number of patterns after generation. Higher values instruct the tool to work harder at compacting the pattern set, potentially reducing pattern count at the cost of longer run time. In practice, values between 10 and 50 are commonly used in industrial flows as a balance between run time and pattern count reduction.

    ```
    create_patterns -compaction_effort 15
    ```

  – **-pattern_limit**: sets a hard cap on the maximum number of patterns the tool is allowed to generate. This is useful in flows where test time on the ATE is strictly constrained.

```
create_patterns -pattern_limit 50000
```

create_patterns also fully supports multiprocessing, both through multithreading on a local machine and through distributed processing across multiple remote hosts. This allows the ATPG computation to be parallelized and can significantly reduce run time on large designs. Multiprocessing is configured through the *add_processors* and *set_multiprocessing_options* commands before invoking *create_patterns*:

```
set_multiprocessing_options -multithreading on -optimize_secondaries balance
add_processors localhost:8
add_processors lsf:48
```

The -optimize_secondaries option accepts three strategies:

- **pattern_count** (default): focuses on keeping the pattern set as compact as possible.
- **runtime**: focuses on finishing as fast as possible, which is useful during early development iterations.
- **balanced**: a compromise between the two, balancing pattern count and run time.

- **report_statistics** displays the final coverage report, showing how many faults were detected, the achieved test coverage percentage, and a breakdown by fault class. This is one of the most important outputs of any ATPG run:

```
report_statistics
```

- **write_patterns** saves the generated test patterns to a file in a format suitable for loading onto an ATE or for use in downstream simulation. Common output formats include STIL and WGL:

```
write_patterns output_patterns.stil.gz -stil -replace
```

Together, these commands define the skeleton of a typical Tessent ATPG session. In practice, a full run script may contain dozens of additional configuration commands, but the sequence above captures the essential logical flow that is reflected in every ATPG log file — and that the ATPG Run Analyser is built around parsing and interpreting.

3.2 EDT and scan compression

Embedded Deterministic Test, or EDT, is a scan compression technique used to reduce the amount of test data that must be handled during ATPG. As shown in the figure, the ATE does not directly access all internal scan chains of the chip. Instead, it sends a smaller number of compressed patterns through a limited number of scan channel inputs. Inside the chip, a decompressor expands these compressed values and distributes them to the many internal scan chains of the core design. In this way, a small external interface can control a much larger internal scan structure.

At the same time, the responses captured from the scan chains are not sent back to the tester directly in full. They are collected by a compactor, which combines the internal responses into a smaller number of compressed expected responses that can be compared by the ATE. This reduces the amount of data that must be transferred out of the chip and helps to lower both test time and tester memory requirements. In practice, EDT makes it possible to use many internal scan chains while keeping the external test interface much smaller and more efficient.

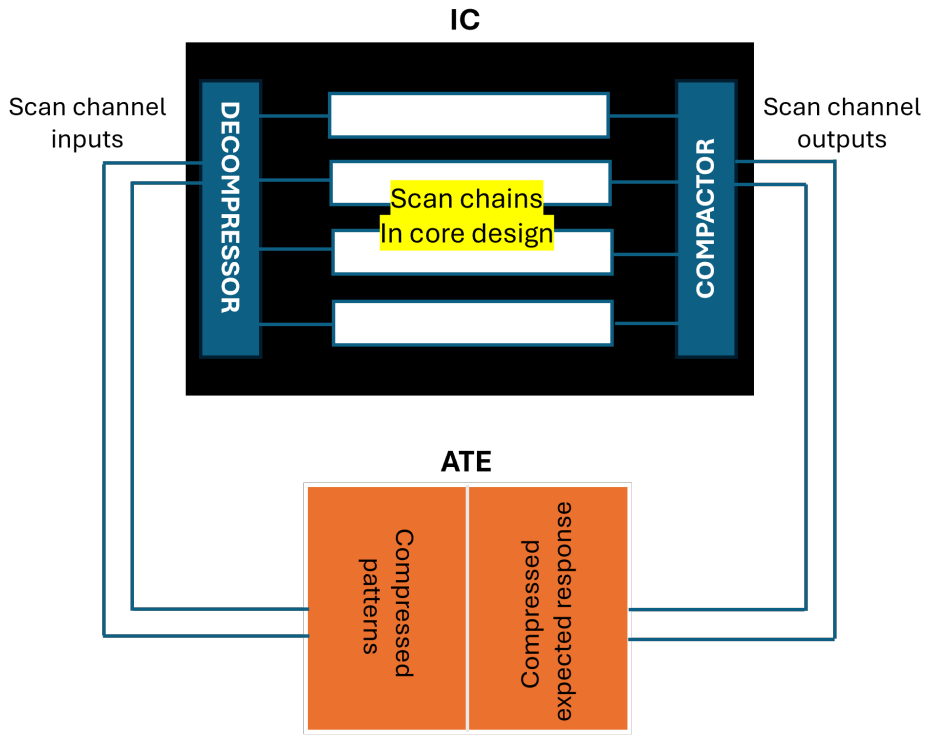


Figure 3.1: EDT scan compression architecture

3.3 Test patterns generation with Siemens Tessent

An ATPG run is not a single monolithic operation. It is a structured sequence of phases, each with a distinct purpose, that together transform a gate-level netlist into a set of test patterns

ready for use in manufacturing tests. Understanding this flow is essential to appreciate why profiling and monitoring such a run is both useful and non-trivial.

The basic process flow for an ATPG tool is shown in the following figure.

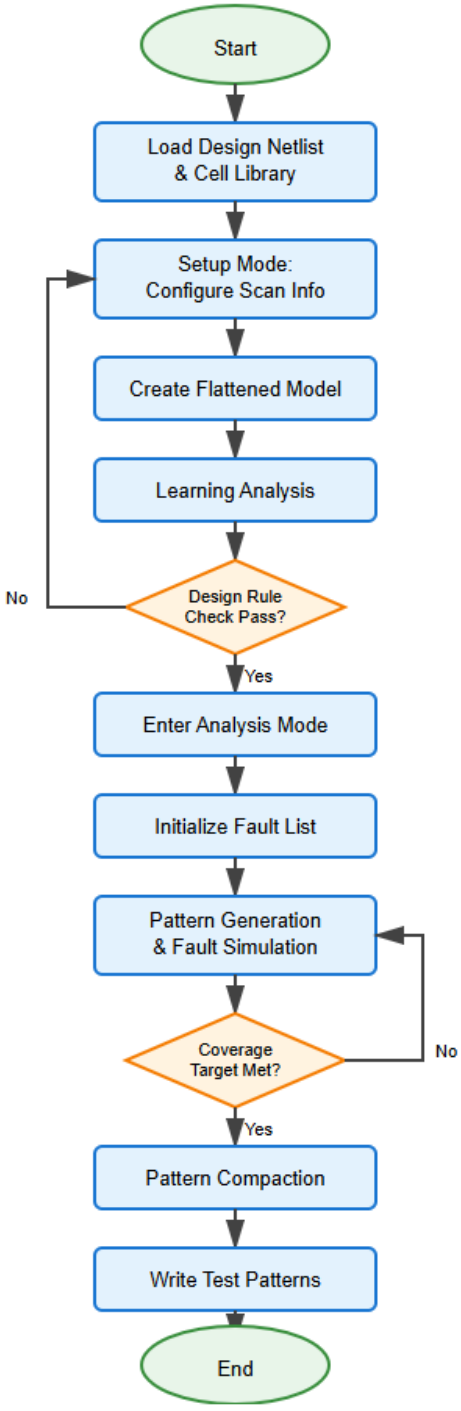


Figure 3.2: ATPG process flow

The ATPG flow in Tessent starts by launching Tessent Shell with the command *tessent -shell* and setting the context to patterns-scan using *set_context*, which enables the ATPG functionality. After that, the main design inputs must be loaded. These are the gate-level netlist and the DFT library, which are read with the *read_verilog* and *read_cell_library*

commands. Tessent checks that the elements in the netlist are correctly described in the library before continuing.

Once the design and library have been loaded, the tool enters setup mode. In this phase, the user can configure the design and the scan information, either by typing commands directly or by using a dofile. Other settings that affect the flattening of the design and the creation of the simulation model can also be specified here. When the setup is complete, exiting setup mode triggers the creation of a flattened model of the design, if it does not already exist. Tessent then performs learning analysis on this model to better understand the circuit behaviour before starting ATPG.

After learning analysis, the tool runs design rule checking to verify that the design is ready for test generation. If the design passes this step, Tessent enters analysis mode. In this mode, the user can perform fault simulation and prepare the fault list for ATPG. At this point, the actual pattern generation can begin. This is usually done in one or more passes, where each pass applies the ATPG algorithm to the current fault list and tries to detect as many remaining faults as possible.

The first pass is often a relatively fast sweep that focuses on the easiest faults. Its main purpose is to establish an initial level of coverage before moving on to more difficult faults. In many cases, the patterns produced in this phase are mainly used as a starting point for later passes, rather than as the final output.

Each pass works iteratively. The ATPG engine selects an undetected fault, tries to generate a test pattern for it, and either adds the pattern to the set or marks the fault as untestable if no pattern can be found. This process continues until all faults have been processed or until a stopping criterion is reached.

These stopping criteria are important because they represent one of the main tradeoffs in ATPG. Running longer can improve coverage, but it also increases computation time and usually leads to a larger pattern set, which in turn increases manufacturing test time.

During ATPG, the tool generates test patterns and fault-simulates them to verify which faults are detected. Depending on the design and the test goals, ATPG can be run with different settings, and additional patterns can be added to improve the final test coverage.

After the initial passes, the achieved fault coverage may still be below the target set by the design team. To improve the result, the tool may run additional pattern generation phases focused on the remaining undetected faults.

This stage usually applies more effort to the hardest faults and may use different settings to push coverage higher. Because it targets the most difficult parts of the design, this phase is often the most time-consuming part of the entire run.

In practice, multiple refinement iterations may be needed. Each new iteration usually gives a smaller benefit than the previous one, while requiring a similar or even greater amount of computation time. For this reason, engineers must decide when the extra coverage no longer justifies the additional runtime.

The metrics produced during these iterations, such as the number of generated patterns,

the coverage reached, and the CPU time consumed, are very useful for understanding the behaviour of a run and comparing it with a reference baseline.

Once pattern generation is complete, the tool may perform a compaction step to reduce the total number of patterns. The idea is to combine patterns where possible, so that the final set is smaller and more efficient to apply during manufacturing test. This is important because the number of patterns directly affects test time, which has a strong impact on the overall cost of production.

In some cases, patterns coming from different passes or refinement stages may also be merged into a single final set. The result is the pattern set that will later be used for test application.

An ATPG run can take a significant amount of time, especially for large designs. The total runtime depends on the size of the circuit, the target coverage, and the tool settings used during the run. For this reason, ATPG is one of the most computationally expensive steps in the test flow.

There is always a tradeoff between runtime and output quality. If the tool is allowed to spend more time on each fault, it may achieve higher coverage, but the run will be slower and the final pattern set may become larger. On the other hand, a faster run may finish sooner, but it can leave more faults undetected. Understanding these tradeoffs is important when analysing and comparing different runs, since a change in runtime does not always mean a problem if the configuration has also changed.

The ATPG tool works with several inputs to generate test patterns and the related analysis files. The main input is the design itself, which must be provided in gate-level Verilog format. In addition, the tool requires the design library, which contains the cell descriptions needed to build the flattened simulation model used during fault simulation and pattern generation. Another important input is the test procedure file, which defines how the scan circuitry should operate in the design. This file can be written manually, or it can be generated automatically by Tessent Scan using the *write_atpg_setup* command. In some cases, ATPG can also start from a previously saved fault list, which gives the tool a set of faults and their current status before pattern generation begins.

From these inputs, ATPG produces several outputs. The most important output is the set of test patterns, which can be written in different simulator or ASIC vendor formats depending on the target flow. In addition to the patterns, the tool also generates ATPG information files, such as the log file of the session, which record useful information about the run. Finally, ATPG can write out a fault list in an ASCII-readable format, containing the internal fault information maintained by the tool.

3.4 Analysis of *report_statistics*

The *report_statistics* command gives a final summary of an ATPG run. It shows how the faults are classified, the coverage reached, the number of generated patterns, and the total CPU time

used by the tool. This makes it one of the most useful outputs in a Tessent run, because it gives a quick view of both the quality of the test and the cost of the computation.

From the report, the main fault classes can be observed. Some faults are detected successfully, while others remain undetected because they are not controllable, not observable, or not suitable for test generation under the current constraints. Among the reported classes, RE stands for redundant faults. These are faults that belong to logic which does not affect the final behaviour of the circuit, so even if the fault is present, it cannot be observed at the outputs. In this sense, redundant faults are related to redundant logic in the design, meaning logic that is structurally present but does not contribute to the functional behaviour of the circuit. Since these faults cannot be detected in practice, they are usually excluded from the calculation of test coverage.

The report also includes the final test coverage, fault coverage, and ATPG effectiveness. These values are important because they give a compact view of the overall quality of the run. In industrial flows, they are often compared with a reference or golden run to check whether the current result is aligned with expectations. For this reason, *report_statistics* is not only useful for final reporting, but also for monitoring and comparing ATPG runs over time.

3.5 Tessent ATPG log files

Every execution of an ATPG tool produces a log file. This file is the primary record of everything that happened during the run: which phases were executed, when each phase started and ended, what metrics were achieved at each step, and whether any errors occurred. For a run that lasts many hours, the log file can grow to a very large size. Understanding the structure and content of this file is the starting point for any automated analysis tool.

3.5.1 Structure and format

ATPG log files are plain text files, written line by line as the tool executes. Each line typically contains a timestamp, an event type or category identifier, and a message describing what happened. The timestamp is recorded at the moment the line is written, which means that the spacing between consecutive lines reflects the actual time taken by the operations between them. A dense cluster of lines with closely spaced timestamps indicates a fast-moving phase, while a long gap between two lines indicates that the tool was busy with a computationally intensive operation.

The timestamp format is standardised within a given tool family and follows a fixed pattern that includes the year, month, day, hour, minute, and second. This level of precision is sufficient for profiling purposes, since the operations of interest, such as ATPG phases and pattern generation iterations, typically last from seconds to hours. The inclusion of the full date, rather than just the time of day, is important for runs that span midnight or cross month boundaries, as naive time arithmetic that ignores the date would produce incorrect duration

calculations in these cases.

Beyond timestamps, log lines carry structured markers that identify the beginning and end of specific phases and tasks. These markers follow a consistent naming convention that allows a parser to identify, for example, when the tool transitions from the design loading phase to the first pattern generation pass, or when a refinement iteration begins and ends. The presence of these markers is what makes automated log parsing feasible, because the parser can look for specific patterns instead of trying to interpret free-form text.

3.5.2 Per-iteration metrics

In addition to phase boundary markers, the log file contains detailed per-iteration metrics for the most computationally intensive operations. During pattern generation, for example, the tool writes a summary line at the end of each iteration that records the number of patterns generated so far, the current test coverage, the current fault coverage, and the CPU time consumed. These lines form a time series that captures the progression of the run in fine-grained detail.

This per-iteration data is extremely valuable for profiling and comparison purposes. By plotting test coverage as a function of time, an engineer can see immediately how quickly the run is converging toward the coverage target, and whether the convergence rate is consistent with previous runs on the same design. A run that is converging more slowly than expected may indicate a change in the design, a change in the tool configuration, or a performance issue in the computing environment.

The per-iteration metrics also capture information about the options used for each invocation of the pattern generation command, such as the coverage effort level and the abort limit in effect at that point.

The quality and robustness of the parsing logic are therefore critical to the reliability of the entire tool. A parser that misidentifies a phase boundary, misreads a timestamp, or fails to handle an unexpected log format will silently produce incorrect results that may mislead the engineer. This is one of the main motivations for the extensive unit testing described in the previous chapter: every parsing function is tested against carefully constructed log snippets that cover both normal cases and edge cases, ensuring that the parser behaves correctly across the full range of inputs it may encounter in practice.

Chapter 4

Software Quality and Unit Testing

While previous chapters focused on the Design-for-Test methodology and the Tessent ATPG tool, this thesis also represents a substantial software engineering effort. The ATPG Run Analyzer is a tool that reads industrial log files containing thousands of lines of data, extracts meaningful information through complex parsing algorithms, generates multi-format reports, and sends automated notifications. Given the critical nature of these operations in a semiconductor testing workflow, software quality becomes a paramount concern.

If a parsing function returns incorrect values, if metric calculations contain subtle bugs, or if monitoring alerts fail to trigger, engineers may make decisions based on erroneous information. In an industrial setting where ATPG runs consume significant computational resources and time (often 2-12 hours per run), incorrect analysis could lead to wasted effort, missed design defects, or false confidence in test quality. For these reasons, correctness, reliability, and maintainability are essential attributes of the tool.

To ensure these quality attributes, the development process incorporated Test-Driven Development (TDD) principles and implemented a comprehensive unit testing strategy. This approach treats automated testing not as an afterthought, but as an integral part of the development workflow, where tests are written alongside—or even before—the production code they verify.

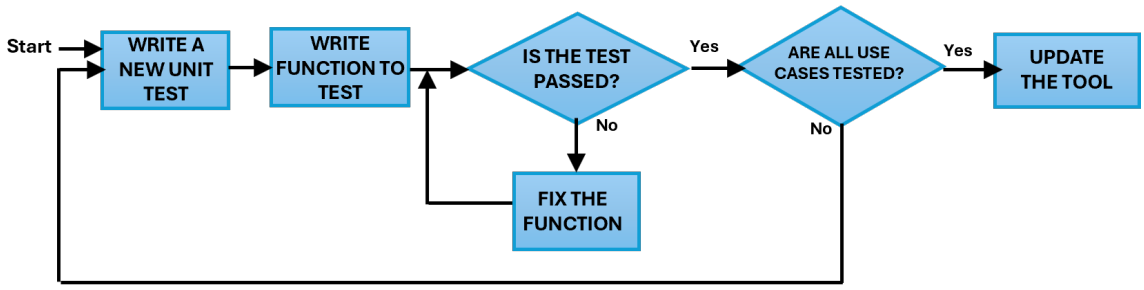


Figure 4.1: Unit testing scheme

4.1 Test-Driven Development Methodology

Test-Driven Development is a software engineering practice where tests are written before implementing the functionality they verify. The typical TDD cycle, often called ”Red-Green-Refactor,” consists of three phases:

1. **Red Phase**: Write a test for a new feature or function. The test fails initially because the functionality does not yet exist.
2. **Green Phase**: Implement the minimum code necessary to make the test pass.
3. **Refactor Phase**: Improve the code structure while ensuring all tests continue to pass.

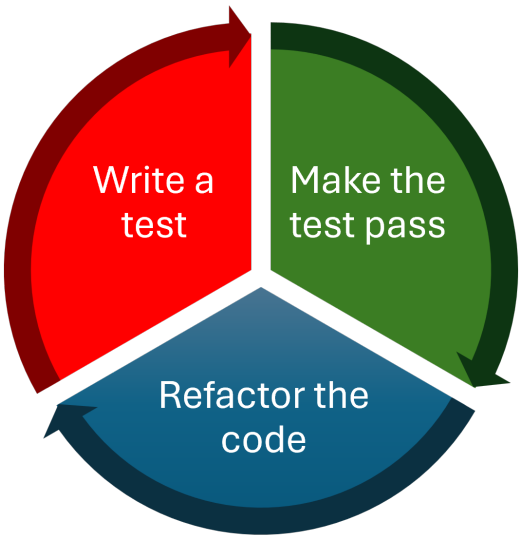


Figure 4.2: Test-Driven Development cycle

This methodology was adopted for the ATPG Run Analyzer project, particularly for critical components such as time calculations, log parsing, and metric extraction. The benefits of this approach manifested in several ways throughout the development process:

Early Bug Detection: By writing tests first, the expected behavior is explicitly defined before implementation begins. This prevents ambiguity about what constitutes correct behavior and catches logic errors immediately when the code is written, rather than weeks later during system testing.

Design Improvement: Writing tests before code forces consideration of the function’s interface and usage patterns. This leads to cleaner, more modular designs because functions must be testable in isolation. For instance, the time calculation functions in the ATPG Run Analyzer were designed to accept explicit year, month, and day parameters rather than relying on global state, making them both easier to test and more reusable.

Regression Prevention: The comprehensive test suite created through TDD serves as a safety net for future modifications. When new features are added or bugs are fixed, the

existing test suite verifies that no previously working functionality has been broken. This was particularly valuable during the project when timestamp handling was extended to support year and month boundaries—the existing test suite immediately caught several edge cases that would have been difficult to discover through manual testing.

Living Documentation: Tests written in the TDD style serve as executable documentation. By reading the tests, another developer (or the original developer months later) can understand what each function is supposed to do, what inputs it accepts, and what outputs it produces. This proved invaluable when extending the monitoring functionality to include coverage alerts, as the existing test structure provided clear patterns to follow.

4.2 Python unittest

The testing framework used in this project is Python's built-in *unittest* module. This choice is practical because it is part of the standard Python library and provides all the basic tools needed to write organised tests. The tests are grouped into classes that inherit from *unittest.TestCase*. Each class usually covers one part of the application, such as utility functions, profiling functions, or monitoring functions.

Each test method starts with the prefix *test_*, which allows the framework to find and run it automatically. This makes the test suite easy to maintain and extend. The framework also supports two useful methods, *setUp()* and *tearDown()*. The *setUp()* method runs before each test and is used to prepare common data, while *tearDown()* runs after each test and is used to clean up temporary files or reset the test environment. This keeps the tests independent from each other.

The main way to check results in unittest is through assertions. These are small checks that compare the actual result of a function with the expected one. Some of the most common assertions used in this project are:

- *assertEqual(a, b)*, which checks that two values are the same.
- *assertTrue(x)*, which checks that a condition is true.
- *assertFalse(x)*, which checks that a condition is false.
- *assertIn(a, b)*, which checks that a value is present in a string, list, or other container.
- *assertRaises()*, which checks that a function raises the expected exception.

For example, *assertEqual()* can be used to verify that a function returns the correct number of detected faults. *assertTrue()* can be used to check that a condition is satisfied, such as the presence of a task in a parsed log. *assertIn()* is useful when checking whether a specific keyword appears inside a string or a report. *assertRaises()* is useful when testing error handling, for example when invalid input is given to a function.

4.3 Structure of the test suite

The test suite of the ATPG Run Analyser is divided into three main files:

Tests are organized into classes that are inherited from *unittest.TestCase*. Each class typically covers a cohesive module or subsystem of the application. In the ATPG Run Analyzer, the test suite comprises four main test modules:

- **test_utils.py (12 tests)**: Covers shared utility functions including time calculations, email formatting, log validation, and data structure builders
- **test_profiler.py (21 tests)**: Verifies profiling logic including log parsing, metric extraction, Excel report generation, and comparison operations
- **test_summary_generation.py (8 tests)**: Tests HTML report generation, status display formatting, and incomplete run handling
- **test_monitor.py (12 tests)**: Validates monitoring logic, alert generation, task detection, and coverage checking

This separation makes the tests easier to understand and maintain. Each file corresponds to a specific part of the application, so it is easier to locate the relevant tests when a function changes. This also makes the project more organized, because the test structure follows the structure of the source code.

The following sections present examples of different testing strategies used in the project. The code examples shown illustrate typical testing patterns and approaches, with function names and implementations simplified for clarity. These are representative examples that demonstrate the testing methodology, rather than the actual production code from the ATPG Run Analyzer.

4.4 Testing pure functions

Some of the simplest tests are those that verify pure functions. These are functions that take inputs, perform a calculation, and return a result without interacting with the outside world. In this project, several utility functions fall into this category, especially those related to time calculations.

The tool needs to compute elapsed time between timestamps extracted from log files. This is not always as simple as it seems, because log files can span multiple days, cross month boundaries, or even cross year boundaries. A naive implementation that only subtracts day numbers would give incorrect results in cases such as a run starting on January 31st and ending on February 1st. For this reason, the calculation must be calendar-aware and handle differences in month length and leap years correctly.

Testing this kind of function is straightforward. The test provides known input values, such as a start date, an end date, and the corresponding time components, and then checks whether the function returns the expected elapsed time in seconds. Since the expected value can be calculated in advance, the test is completely deterministic and easy to validate.

For example, consider a function that calculates the absolute timestamp in seconds from the start of a run. A typical test case for crossing month boundaries would look like this:

```
def test_timestamp_crossing_month_boundary(self):
    """Test timestamp calculation when crossing from January to February"""
    start_year = 2026
    start_month = 1
    start_day = 31

    # Event occurs on February 1st at 02:30:00
    event_month = 2
    event_day = 1
    event_hour = 2
    event_minute = 30
    event_second = 0

    # Expected: 1 full day (86400 sec) + 2.5 hours (9000 sec) = 95400 sec
    expected_timestamp = 95400

    result = calculate_absolute_timestamp(
        start_year, start_month, start_day,
        event_month, event_day, event_hour, event_minute, event_second
    )

    self.assertEqual(result, expected_timestamp)
```

Similar tests are used for functions that compute absolute timestamps relative to the first event in a log, calculate the duration of individual tasks, or express a task duration as a percentage of the total run time. These functions form the numerical basis of the reporting logic, so their correctness is essential. If one of them is wrong, every report produced by the tool could be affected.

Another example tests the percentage calculation function:

```
def test_calculate_percentage(self):
    """Test percentage calculation for task duration"""
    task_duration = 3600 # 1 hour in seconds
    total_duration = 7200 # 2 hours in seconds

    result = calculate_percentage(task_duration, total_duration)

```

```
    self.assertEqual(result, 50.0)
```

4.5 Testing file I/O

A large part of the ATPG Run Analyser works with files. The tool reads log files, extracts information from them, and writes reports and other output files. Testing this type of functionality requires a different approach from testing pure functions, because the tests need to interact with the file system in a controlled and repeatable way.

For this reason, the *tempfile* module is used throughout the test suite. This module allows the creation of temporary files and directories that exist only for the duration of the test. A test can create a temporary file, write a small and carefully chosen piece of log content into it, call the function under test, and then check whether the result is correct. After the test ends, the temporary file is removed automatically, so it does not leave any trace on the file system.

This approach has several advantages. First, the test is self-contained, because it does not depend on any external file being available in a specific location. Second, the input is fully controlled, which makes it easier to test specific scenarios and edge cases. Third, the test is reproducible, since the same input always produces the same result.

Here is an example of a test that verifies the log parsing function can correctly extract phase boundaries from a log file:

```
import tempfile
import unittest

class TestLogParsing(unittest.TestCase):
    def test_extract_phase_boundaries(self):
        """Test extraction of phase start and end markers from log"""
        # Create temporary log file with sample content
        log_content = """\
(I): some line @ [2026/04/22 03:34:48]
(I): task_break, ::atpg::task_break early_phase @ [2026/04/22 03:34:50]
(I): some line @ [2026/04/22 05:12:33]
(I): task_break, ::atpg::task_break main_phase @ [2026/04/22 05:12:33]
(I): some line @ [2026/04/22 15:45:20]
(I): task_break, ::atpg::task_break post_phase @ [2026/04/22 15:45:20]
"""
        with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
            f.write(log_content)
            temp_log_path = f.name

        # Call the function under test
        phases = extract_phase_boundaries(temp_log_path)
```

```
22
23          # Verify the extracted phases
24          self.assertEqual(len(phases), 3)
25          self.assertEqual(phases[0]['name'], 'early_phase')
26          self.assertEqual(phases[0]['hour'], 3)
27          self.assertEqual(phases[0]['minutes'], 34)
28          self.assertEqual(phases[0]['seconds'], 50)
29          # Cleanup
30          os.unlink(temp_log_path)
```

The same idea is also used for output files. For example, when a function generates a CSV file or an Excel report, the file is written to a temporary location and then read back to verify its contents. This gives confidence that the complete input-output chain is working correctly, from reading the log data to producing the final output.

An example test for CSV file generation:

```
1  def test_generate_task_duration_csv(self):
2      """Test CSV generation for task durations"""
3      task_data = {
4          'early_phase': 3600,
5          'main_phase': 36000,
6          'post_phase': 18000
7      }
8
9      with tempfile.TemporaryDirectory() as tmpdir:
10         output_path = os.path.join(tmpdir, 'task_duration.csv')
11
12         # Generate CSV file
13         generate_task_duration_csv(task_data, output_path)
14
15         # Verify file exists and contains correct data
16         self.assertTrue(os.path.exists(output_path))
17
18         df = pd.read_csv(output_path)
19         self.assertEqual(len(df), 3)
20         self.assertEqual(df.loc[0, 'task_name'], 'early_phase')
21         self.assertEqual(df.loc[0, 'duration_seconds'], 3600)
```

4.6 Regression testing with reference files

For the most complex outputs, such as multi-sheet Excel workbooks and HTML email summaries, a more advanced testing strategy is needed. In these cases, the project uses regression

testing against reference files.

The idea is simple. At a point in time when the tool is known to produce correct output, the generated files are saved as reference artefacts. These files are kept in a dedicated test directory. Later, when the tool is run again with the same input, the new output is compared against the stored reference.

If the output matches the reference exactly, the test passes. If anything differs, the test fails and reports the discrepancy. This makes regression testing very useful for detecting unintended changes. Even a small modification in the parsing logic can affect the final report, so comparing against a well-known result helps catch these issues early.

In this project, the comparison is done with the help of the *pandas* library, which makes it possible to check data in a structured way. This is especially useful for Excel-based outputs, where the content must remain consistent over time.

Here is an example of a regression test for Excel report generation:

```
import pandas as pd
import unittest

class TestExcelReports(unittest.TestCase):
    def test_excel_report_regression(self):
        """Test that Excel report matches reference output"""
        # Path to reference file generated from known-good run
        reference_path = 'tests/reference_files/profiler_report_ref.xlsx'

        # Generate new report from same input
        with tempfile.TemporaryDirectory() as tmpdir:
            output_path = os.path.join(tmpdir, 'test_report.xlsx')

            # Use same test log file used to create reference
            generate_profiler_excel(
                log_path='tests/fixtures/sample_log.txt',
                output_path=output_path
            )

            # Load both files and compare
            ref_df = pd.read_excel(reference_path, sheet_name='Summary')
            new_df = pd.read_excel(output_path, sheet_name='Summary')

            # Compare dataframes
            pd.testing.assert_frame_equal(ref_df, new_df)

            # Also check that all expected sheets exist
            ref_sheets = pd.ExcelFile(reference_path).sheet_names
            new_sheets = pd.ExcelFile(output_path).sheet_names
```

```
            self.assertEqual(ref_sheets, new_sheets)
```

Another example shows how to test HTML email summary generation against a reference:

```
def test_html_summary_regression(self):
    """Test that HTML summary matches reference output"""
    reference_html = 'tests/reference_files/email_summary_ref.html'

    # Generate new HTML summary
    html_content = generate_email_summary(
        run_data=self.sample_run_data,
        metrics=self.sample_metrics
    )

    # Load reference HTML
    with open(reference_html, 'r') as f:
        expected_html = f.read()

    # Compare content (normalizing whitespace)
    self.assertEqual(
        ' '.join(html_content.split()),
        ' '.join(expected_html.split())
    )
```

4.7 Using unittest.mock

In addition to the standard unittest assertions, this project also makes use of the *unittest.mock* module. This module is very useful when a test needs to isolate the function being checked and avoid depending on external effects. In practice, it allows real functions or objects to be temporarily replaced by mock versions, so that the behaviour of the code can be tested in a controlled way. This was especially useful in this project because some functions print messages to the terminal, some depend on the current time, and some interact with external services such as email.

One common use of *unittest.mock* in this project was to silence *print()* during testing. The real code produces several status messages, but these are not important for the unit tests and would only make the output harder to read. For this reason, *builtins.print* was patched in the test setup and removed again in the test teardown. This means that the tests can run without showing extra console output, while the original *print()* behaviour is restored after the test class finishes.

Here is an example of mocking print statements:

```
from unittest.mock import patch
```

```
class TestMonitor(unittest.TestCase):
    def setUp(self):
        # Mock print to suppress output during tests
        self.print_patcher = patch('builtins.print')
        self.mock_print = self.print_patcher.start()

    def tearDown(self):
        # Restore original print function
        self.print_patcher.stop()

    def test_monitor_stuck_detection(self):
        """Test that stuck run detection works correctly"""
        # Function will call print() internally, but it's mocked
        result = detect_stuck_run(log_path='test.log', prev_line_count=100)

        # Verify the logic without seeing print output
        self.assertTrue(result['is_stuck'])
        self.assertEqual(result['stuck_duration'], 7200)
```

Another important use of mocking was for time-dependent functions. For example, the function that calculates the elapsed time since the ATPG bootup event depends on the current system time. Since the real clock changes all the time, the test would not give the same result every time it is run. To avoid this problem, *datetime* was mocked so that the test could use a fixed time value. In this way, the function could always be checked against the same expected result, making the test stable and repeatable.

An example of mocking datetime for time-dependent tests:

```
from unittest.mock import patch, MagicMock
from datetime import datetime

class TestTimeCalculations(unittest.TestCase):
    @patch('utils.datetime')
    def test_elapsed_time_since_bootup(self, mock_datetime):
        """Test elapsed time calculation with fixed current time"""
        # Set a fixed "current time" for the test
        mock_now = datetime(2026, 4, 22, 10, 30, 0)
        mock_datetime.now.return_value = mock_now

        # Bootup was at 2026/04/22 08:00:00
        bootup_time = datetime(2026, 4, 22, 8, 0, 0)

        result = calculate_elapsed_since_bootup(bootup_time)
```

```
16
17          # Expected: 2.5 hours = 9000 seconds
18          self.assertEqual(result, 9000)
```

Mocking was also used when testing email-related functions. Instead of connecting to a real SMTP server and sending an actual email, the SMTP object was replaced by a mock version. This allowed the test to check whether the email function was called correctly, without sending anything outside the machine. In these tests, the mock object was also inspected after the call. The attribute *mock.called* was used to verify that the method was invoked, while *mock.call_args* was used to check the arguments passed to it, such as the sender, receiver, and message content. This is important because it allows the test to confirm that the function behaves correctly without relying on external infrastructure.

Here is an example of mocking email sending:

```
1  from unittest.mock import patch, MagicMock
2
3  class TestEmailAlerts(unittest.TestCase):
4      @patch('smtplib.SMTP')
5      def test_send_alert_email(self, mock_smtp):
6          """Test that alert emails are sent with correct content"""
7          # Create mock SMTP instance
8          mock_smtp_instance = MagicMock()
9          mock_smtp.return_value = mock_smtp_instance
10
11         # Call function that sends email
12         send_stuck_alert(
13             recipient='engineer@company.com',
14             log_path='/path/to/log.txt',
15             phase='main_phase',
16             duration='02:00:00'
17         )
18
19         # Verify SMTP was called
20         self.assertTrue(mock_smtp.called)
21         self.assertTrue(mock_smtp_instance.sendmail.called)
22
23         # Check email arguments
24         call_args = mock_smtp_instance.sendmail.call_args
25         sender, recipient, message = call_args[0]
26
27         self.assertEqual(recipient, 'engineer@company.com')
28         self.assertIn('stuck', message.lower())
29         self.assertIn('main_phase', message)
```

```
30          self.assertIn('02:00:00', message)
```

Overall, *unittest.mock* made the test suite cleaner and more reliable. It allowed the tests to focus only on the logic of the code, while external dependencies such as console output, system time, and email sending were safely controlled. This was especially important for the ATPG Run Analyser, because the tool includes several functions that need to behave consistently under different conditions.

4.8 Why this testing approach is useful

This testing strategy is well suited to the ATPG Run Analyser because the tool performs several different tasks that must all work correctly. It reads log files, extracts statistics, generates reports, compares runs, and sends alerts. If even one of these parts is incorrect, the final result may be unreliable. By testing each component separately, the risk of hidden bugs is reduced, and the overall quality of the tool is improved.

Another advantage of this approach is that it supports future development. When new features are added or existing functions are modified, the test suite can be run again to check that nothing has been broken. In this way, unit testing works as a safety net that helps keep the code stable over time.

Chapter 5

ATPG Run Analyzer — Overview and Architecture

5.1 System architecture

The ATPG Run Analyzer is a Python-based command-line tool developed to support DFT engineers in the analysis and monitoring of ATPG runs. As discussed in the previous chapters, these runs can last many hours and produce large, detailed log files that are not practical to inspect manually. The tool addresses this by automating the extraction, analysis, and reporting of the most relevant information from those files.

The tool is built around two distinct operating modes, each designed for a different moment in the ATPG workflow. The Profiler mode handles completed runs, performing a thorough analysis of the log file and generating structured reports. The Monitor mode is designed for ongoing runs, continuously watching the log file and sending alerts when something looks wrong. Both modes share a common reference management system, which allows engineers to create and use good run baselines for comparison and anomaly detection.

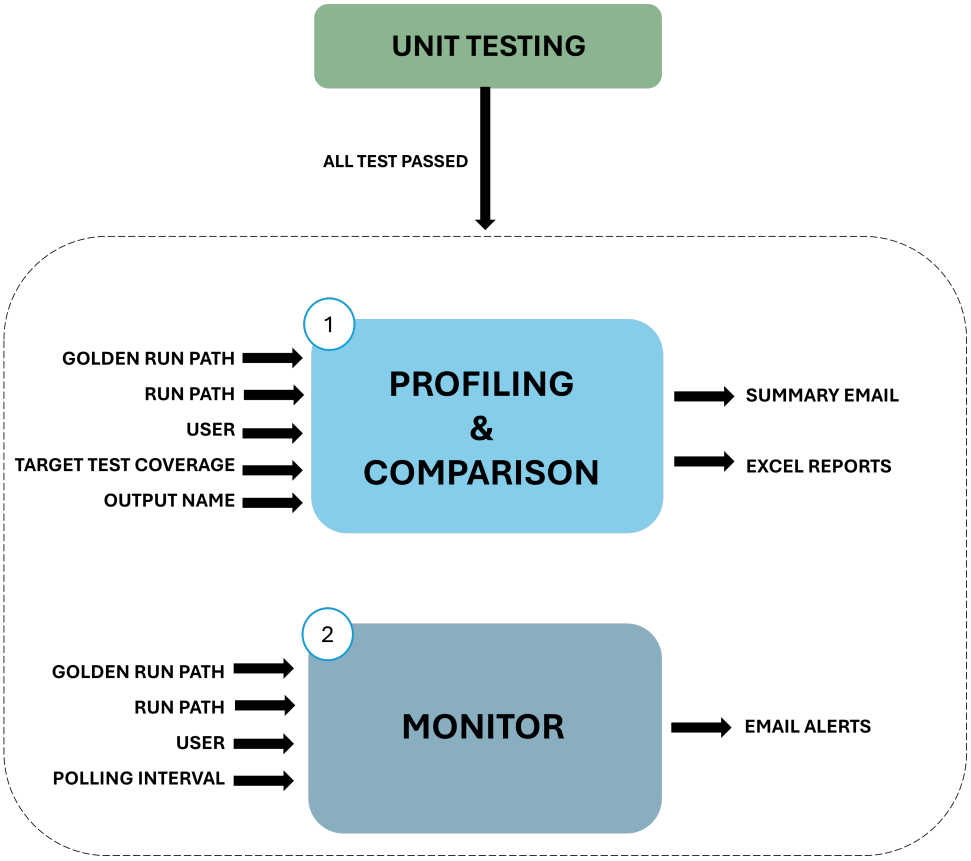


Figure 5.1: System Architecture Overview

5.1.1 High-level design

The tool follows a modular architecture where each operational mode is implemented as an independent component, but both share a common set of utility functions. This design keeps the code well organized and avoids duplication, since operations like log parsing, time calculations, and email sending are used across all modes and are therefore centralized in a shared utilities module.

Profiler and Monitor communicate with the shared utilities layer but remain independent of each other. This means each mode can be developed, tested, and updated without affecting the others.

5.1.2 Technology stack

The tool is built entirely in Python 3.6 or higher. The main libraries used are:

- `pandas`, for reading and manipulating CSV and Excel data
- `openpyxl`, for generating multi-sheet Excel reports with formatting and embedded charts
- `matplotlib`, for generating coverage plots as PNG images

- `smtplib`, for sending email alerts and summary reports
- `unittest`, for the automated test suite that runs at every startup

All these libraries are either part of the Python standard library or widely available, which means the tool can be set up quickly in any standard engineering environment.

5.1.3 Module structure

The project is organized into the following structure:

```
ATPG-Run-Analyzer/
│
├── run.py                          # Main entry point - mode selection and execution
├── utils.py                        # Shared utilities (regex, time, email, validation)
├── config.json                     # Configuration (log paths, references, output name)
│
├── Profiler/                       # Profiler Mode - Post-run analysis
│   ├── profiler.py                 # Main orchestration and workflow
│   ├── profiling_functions.py      # Core parsing and analysis logic
│   ├── summary_generation.py       # HTML email report generation
│   └── tool_excel_reports/         # Generated Excel reports (auto-created)
│       ├── DDMMYY_HHMM_*.xlsx      # Regular profiling reports
│       └── DDMMYY_HHMM_CMP_*.xlsx  # Comparison reports
│
├── Monitor/                        # Monitor Mode - Real-time monitoring
│   ├── monitor.py                  # Main monitoring loop
│   ├── monitor_functions.py        # Alert logic and helper functions
│   └── log_monitor/                # Monitor session logs (auto-created)
│       └── DDMMYY_HHMM_Monitor_*.log
│
├── UnitTest/                       # Test suite (69 tests)
│   ├── test_utils.py               # Tests for utils.py (12 tests)
│   ├── test_profiling_functions.py # Tests for profiling (21 tests)
│   ├── test_monitor_functions.py   # Tests for monitoring (12 tests)
│   ├── test_summary_generation.py  # Tests for HTML generation (8 tests)
│   └── references/                 # Reference files for regression tests
│
│
├── reference_logs/                 # Reference baselines storage
│   └── ref_A/                      # Example reference "ref_A"
│       ├── ref_A_task_duration.csv           # For slow task alerts
│       ├── ref_A_coverage_vs_time.csv        # For coverage alerts & plots
│       ├── ref_A_coverage_vs_time_current.csv # Current run data
│       ├── ref_A_metrics.csv                 # For comparison Excel
│       ├── ref_A_metadata.csv                # Run metadata
│       ├── ref_A_cp_options.csv              # ATPG options
│       └── ref_A_multiprocessing.csv         # Multiprocessing metrics
│
│
│
│
│
├── Plots/                          # Generated coverage plots (auto-created)
│   ├── plot_coverage_vs_patterns.png
│   └── plot_coverage_vs_time.png
│
├── Documentation/
│   └── README.md                   # Main documentation
│
└── Generated Files (runtime):
    └── email_body.html             # HTML email body (generated by Profiler)
```

Figure 5.2: Project Directory Structure

This structure makes it easy to navigate the codebase. Each folder corresponds to a specific part of the tool, and the test files mirror the structure of the source modules, which is a good practice in software development.

5.2 Entry point and startup sequence

When the tool is launched using `run.py`, it does not immediately present the user with a menu. The first and most important step is the automatic execution of the full unit test suite. This is a deliberate design choice: by running all tests before any user interaction, the tool guarantees that its core logic is working correctly in the current environment. The test suite consists of 69 unit tests across four test modules and runs in approximately 0.27 seconds, so the overhead for the user is negligible. If any test fails, execution stops immediately and the user is informed. This means the tool will never silently produce incorrect results because of a broken function.

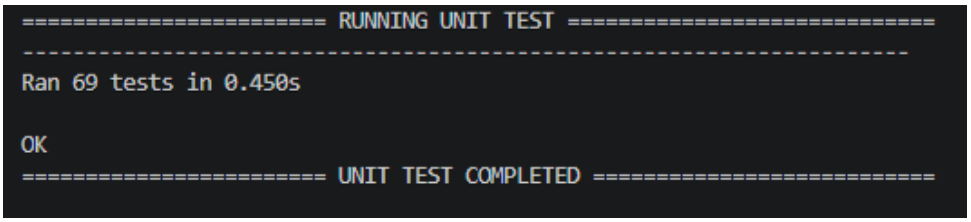


Figure 5.3: Unit Test Execution at Startup

Only after all tests pass does the tool proceed to load the configuration file (`config.json`), which stores reference datasets. Finally, the main menu is displayed and the user can select one of the available modes.

5.3 The main menu

After startup, the user is presented with a simple menu that offers the available modes:

- Profiler
- Monitor

The user selects a mode by entering the corresponding number.



Figure 5.4: Main Menu Interface

The tool also checks the user input carefully. If the user enters an invalid option, the menu is shown again until a valid choice is provided. This keeps the interface simple and avoids unnecessary errors during use.

5.4 Profiler mode

5.4.1 Purpose

The Profiler is the primary analysis mode of the tool. It is designed to process a completed ATPG log file and transform it into structured, human-readable reports. It can also be used on an incomplete log, for example when a run is still in progress and the user wants a partial view of the results so far.



Figure 5.5: Profiler Mode Workflow

5.4.2 User inputs

When the Profiler mode is selected, the user is guided through a short sequence of prompts:

- **Log file path** — the user can choose from a list of pre-configured paths stored in `config.json`, or enter a custom path manually.
- **Output name** — the name for the generated Excel report. A default name is suggested, prefixed with the current date and time.
- **Coverage target** — the minimum test coverage threshold to use in the report. The default value is 96%.
- **Email recipient** — the username of the person who should receive the report. The current system user is offered as the default.
- **Reference selection** — the user can optionally select a reference dataset to compare the current run against. This step can be skipped if no comparison is needed.

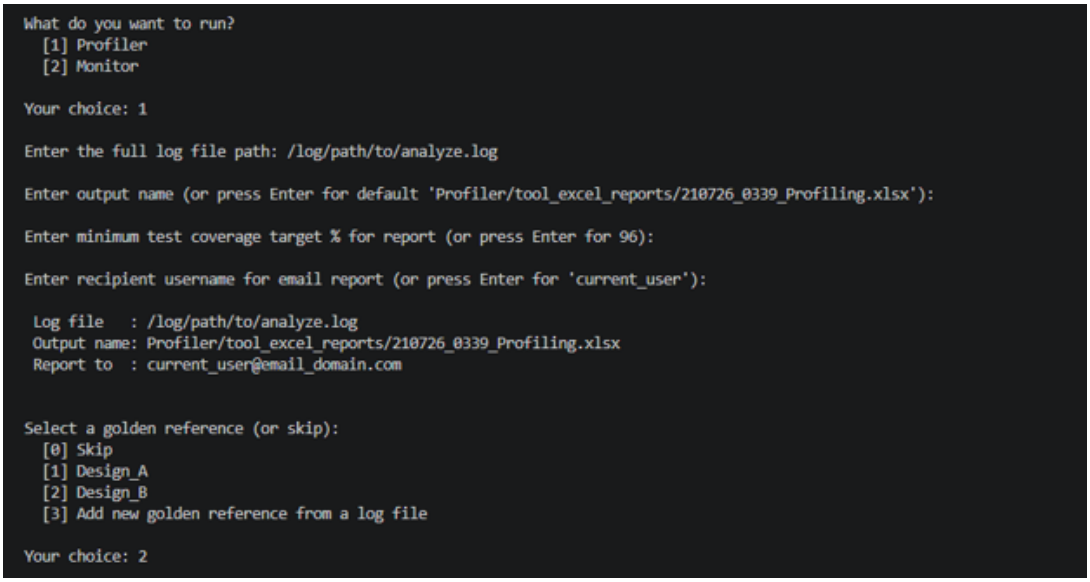


Figure 5.6: Profiler Mode User Prompts

5.4.3 How log parsing works

The first challenge in building the Profiler was figuring out how to extract information from Tessent log files. These logs follow a consistent format with timestamps and structured messages, but there's no formal specification. I had to study actual log files to understand their structure.

The log file contains lines that look like this:

```
[2026/04/22 03:34:48] (I): task_break, ::atpg::task_break early_phase @
[2026/04/22 05:12:33] (I): task_break, ::atpg::task_break late_phase @
```

These are phase boundaries that mark when one major phase ends and another begins. To extract this information, regular expressions (regex) have been used. A regular expression is a pattern that matches specific text. For timestamps, a pattern that captures the year, month, day, hour, minute, and second has been written. For task boundaries, a pattern that captures the phase name has been used.

The parsing strategy reads through the log file line by line. When it finds a timestamp and task boundary, it extracts the phase name and the exact time when that phase started. By finding all phase boundaries in order, it can be determined when each phase began and ended, which lets the tool calculate how long each phase took.

5.4.4 Handling time calculations across calendar boundaries

One of the trickiest problems to solve was calculating elapsed time correctly when ATPG runs span multiple days. If a run starts on April 30 at 11:00 PM and ends on May 1 at 2:00 AM, a naive approach of simply subtracting hours would give completely wrong results.

The solution was to convert all timestamps to an absolute time value — the number of seconds elapsed since the start of the run. For any timestamp found in the log, the tool calculates how many complete days have passed since the run's start date, accounting for different month lengths and leap years, multiplies that by 86,400 (the number of seconds in a day), and adds the time of day in seconds. This gives a monotonically increasing second count that remains correct even across month and year boundaries.

For example, if the run starts on April 30 and a timestamp on May 1 at 2:00 AM is encountered, the tool calculates that 1 complete day has passed (86,400 seconds) and adds 7,200 seconds (2 hours), giving a total elapsed time of 93,600 seconds from the run start.

5.4.5 Extracting coverage and pattern data

Beyond just phase timing, the Profiler also extracts detailed data about pattern generation. The Tessent log contains sections that show iteration-by-iteration progress. The following figure shows an example of such a log section. Note that this is a representative example for illustration purposes and does not correspond to a real production run, as actual run data contains proprietary information.

```
----------------------------------------------------------------------
//  Simulation performed for #gates = 10000000  #faults = 2000000
//  system mode = analysis    pattern source = internal patterns
//  ------------------------------------------------------------------
//  #patterns  test      #faults  #faults    # eff.    # test         process      RE/AU/AAB/EAB
//  simulated  coverage  in list  detected   patterns  patterns       CPU time
//   ---        ------      ---       ---       ---       ---       1200.00 sec     8900/157000/90/0
//   64         12.00%  2000000   152775       61        61       1300.00 sec
//   ---        ------      ---       ---       ---       ---       2300.20 sec     9000/174800/530/1
//  850000 scan cells capture X in 61 out of 64 patterns
//   128        24.00%  1999900   152775       63       124       2400.50 sec
//   ---        ------      ---       ---       ---       ---       3400.00 sec     9001/189000/840/1
//  830000 scan cells capture X in 64 out of 64 patterns
//   192        36.00%  1999000   152775       64       188       3500.00 sec
//   ---        ------      ---       ---       ---       ---       4600.00 sec     9100/200000/1100/2
//  500000 scan cells capture X in 64 out of 64 patterns
//   256        48.00%  1999000   152775       62       250       4820.00 sec
//   ---        ------      ---       ---       ---       ---       5800.30 sec     9160/215000/1400/2
//  760000 scan cells capture X in 62 out of 64 patterns
//   320        58.00%  1990000   152775       64       310       6000.00 sec
//   ---        ------      ---       ---       ---       ---       7100.50 sec     9200/228000/1800/3
//   384        68.00%  1989900   152775       63       370       7200.30 sec
//   ---        ------      ---       ---       ---       ---       8300.00 sec     9300/240000/2200/3
//  840000 scan cells capture X in 64 out of 64 patterns
//   448        70.00%  1988000   152775       64       440       8500.70 sec
//   ---        ------      ---       ---       ---       ---       9600.00 sec     9301/250000/2600/4
//  820000 scan cells capture X in 63 out of 64 patterns
//   512        82.00%  1970000   152775       62       500       9700.50 sec
//   ---        ------      ---       ---       ---       ---      10000.60 sec     9310/260000/3100/4
//   576        85.00%  1900000   152775       64       560      11000.00 sec
//   ---        ------      ---       ---       ---       ---      12000.00 sec     9400/281000/3500/5
//  830000 scan cells capture X in 64 out of 64 patterns
//   640        87.00%  1850000   152775       63       630      12300.00 sec
```

Figure 5.7: Coverage Progress in ATPG Log

These coverage reports appear multiple times throughout the log as ATPG generates more patterns. These lines are parsed using regex to extract the coverage percentages. By collecting

all coverage values along with their timestamps, the tool can track how coverage improved over time.

The pattern generation sections are more complex. They contain tabular data showing patterns generated, coverage achieved, faults detected, and CPU time consumed for each iteration. This data is extracted row by row, storing it in lists that maintain the iteration sequence. Later, this data is used to generate detailed Excel worksheets showing per-iteration metrics.

5.4.6 Analysis process

Once the inputs are collected, the tool analyses the log file in four sequential phases.

The first phase is the **high-level extraction**, where the tool identifies the major execution phases of the ATPG run, extracts their timestamps, and calculates their duration and their percentage of the total run time. This gives the user an immediate overview of how the times are distributed across different stages.

The second phase is the **medium-level analysis**, where the tool goes deeper into each major phase and breaks it down by individual commands. Commands that take more than 10% of their phase's runtime are flagged as heavy operations and get their own dedicated sheet in the Excel report.

The third phase is the **pattern generation analysis**, where the tool extracts the per-iteration data from the pattern generation steps. For each iteration, it records the number of patterns generated, the test coverage reached, the number of faults detected, and the CPU time consumed. This time-series data is very useful for understanding how the run progressed over time.

The fourth phase is the **metrics computation**, where the tool calculates the final summary values: test coverage, fault coverage, total pattern count, and overall timing.

5.4.7 Output structure

The Profiler produces several outputs. In its basic functionality it generates an Excel file and an email with a summary table.

The email summary is a concise HTML report sent directly to the specified recipient. It is designed to give the engineer a complete picture of the run at a glance, without having to open any file. The summary includes:

- the overall run duration and a breakdown of each phase, expressed both in absolute time and as a percentage of the total run
- a detailed analysis of the pattern generation steps, showing how long each step took and how much coverage was gained at each stage
- the final test coverage, fault coverage, and pattern count

- a clear indication of whether the run reached the target coverage or not

If the log file does not contain the termination marker that signals a successfully completed run, a warning is included in the email. If an error line was found in the log, it is also appended to the email body to alert the recipient immediately.

The Excel report is attached to the email and provides a more detailed multi-sheet analysis for engineers who want to dig deeper into the data. Each sheet focuses on a specific aspect of the run:

- a summary sheet with the main metrics and a high-level overview of the run
- a phase breakdown sheet with detailed timing information for every phase of the run
- a pattern generation sheet with per-iteration data, showing how coverage and pattern count evolved over time
- a charts sheet with visual plots of coverage progression and pattern count over time, making it easier to spot trends or anomalies at a glance

The conditional formatting in the report uses colour coding to immediately highlight values that are above or below the expected thresholds, so that the engineer can identify potential problems without reading every number.

If a reference dataset was selected, both the Excel reports and the email summary are extended to include a full comparison between the current run and the reference. This includes a side-by-side table of global metrics such as test coverage, pattern count, and run duration, as well as seven embedded bar charts and two scatter plots showing coverage versus time and coverage versus patterns. Cells where the current run performs worse than the reference are colour-coded in red, while improvements are shown in green.

Finally, two coverage plots are generated as PNG images and embedded inline in the email: one showing test coverage over time, and one showing test coverage versus the number of patterns generated, zoomed between 60 and 100% of test coverage, in logarithmic scale.

5.4.8 Incomplete run handling

The tool handles incomplete log files gracefully. If the log does not contain the termination marker that signals a successfully completed run, a warning is displayed and the run status is shown in red in the email reporting also the last error message encountered in the log file. The tool still extracts all the data that is available and produces a partial report, which can be useful when a run is still in progress or when it was interrupted unexpectedly.

5.5 Monitor mode

Monitor mode is designed for runs that are still in progress. Instead of analyzing a finished log file, the tool repeatedly checks a live log file and looks for signs that the run is progressing

normally. This is useful when a long ATPG job is still running and the engineer wants to know whether it is still active or whether it may have stalled.



Figure 5.8: Monitor Mode Workflow

When this mode is selected, the user is asked to provide:

- the path to the live log file
- the email recipient for alerts
- the polling interval, which defines how often the file should be checked
- a reference dataset, if available, to help determine expected task durations

```
What do you want to run?
  [1] Profiler
  [2] Monitor

Your choice: 2

Enter the log file path: /log/path/to/monitor.log

Enter recipient username for alerts (or press Enter for 'current_user'):

Enter polling interval in minutes (or press Enter for 60):

Select a golden reference for monitoring:
  [0] Enter a custom path
  [1] Design_A
  [2] Design_B
  [3] Add new golden reference from a log file

Your choice: 2

 Log file  :  /log/path/to/monitor.log
 Recipient : current_user@email_domain.com
 Interval  : 60 min
 Reference : Design_B

Monitoring started for:  /log/path/to/monitor.log
```

Figure 5.9: Monitor Mode User Prompts

Monitor mode works flexibly with or without a reference baseline, adapting its behavior based on what's available. This flexibility accommodates different situations engineers encounter in practice.

When a reference baseline is provided, the Monitor performs relative comparisons. It compares the current run's performance against historical data from the golden run, detecting

when tasks take significantly longer than the baseline (1.5× threshold by default) or when test coverage falls below the baseline by a meaningful amount (2 percentage points by default). This approach provides the most sensitive detection because it accounts for the specific characteristics of each design.

When no reference is provided — for example, when working on a completely new design, when ATPG parameters have changed significantly making old baselines invalid, or when simply wanting basic monitoring without the overhead of maintaining baselines — the Monitor uses absolute thresholds that are configured in the system. Tasks that exceed a maximum duration (30 hours by default) trigger slow alerts, and the main pattern generation phase coverage below a minimum threshold (95% by default) triggers quality alerts. These absolute thresholds are configurable through a simple JSON configuration file.

At startup, users are prompted to select a reference baseline from a menu. They can choose an existing reference, create a new one from a completed log, or explicitly skip reference selection. When skipping, the Monitor displays a clear informational message showing exactly what absolute thresholds will be used, ensuring users understand the monitoring behavior.

Importantly, stuck detection works identically regardless of whether a reference is provided, because it doesn't depend on baseline data — it simply checks whether the log file continues growing at each poll. This means even without a reference, engineers get immediate notification if their ATPG job freezes, which is often the most critical type of failure to detect.

The Monitor operates as a continuous loop that runs as long as the ATPG job is executing. Every hour (or whatever interval the user configures), the Monitor wakes up and performs a series of checks on the log file. This polling-based approach is simple and reliable.

Each monitoring cycle follows the same sequence. First, check if the run has completed by looking for the completion marker in the log. If found, stop monitoring. Second, check if the log is ”stuck” by comparing the file size to the previous poll — if it hasn't grown, the ATPG process might have frozen. Third, identify which phase is currently executing by finding the most recent phase boundary marker. Fourth, calculate how long the current phase has been running and compare it to either the baseline (if provided) or the maximum duration threshold (if not). Fifth, check if the main pattern generation phase has completed and if so, verify that coverage meets expectations using either baseline comparison (if provided) or absolute minimum threshold (if not).

5.5.1 Detecting stuck runs

A ”stuck” run means the ATPG process has frozen or crashed but hasn't exited cleanly. The log file stops growing because nothing new is being written. To detect this, the Monitor counts how many lines are in the log file (ignoring empty lines). If the line count is exactly the same across two consecutive polls, it increments a stuck counter. If the counter reaches 2 (meaning the file hasn't changed for 2 hours with default 60-minute polling), the Monitor sends a stuck

alert email.

When the log starts growing again, the counter resets to zero. This allows the Monitor to detect multiple stuck episodes within a single run if the ATPG process repeatedly hangs and resumes. The stuck detection also maintains a flag that tracks whether an alert has already been sent for the current stuck episode. This prevents duplicate alerts from flooding the engineer's inbox if the run stays stuck for many hours.

The stuck alert email includes context to help with debugging: it shows which phase was running when the log stopped growing, how long it's been since the last log update, and any error messages found in the last 20 lines of the log file.

5.5.2 Detecting slow tasks

Even if the ATPG run isn't completely stuck, it might be running much slower than expected. This could indicate resource contention (other jobs competing for CPU), configuration problems, or design issues.

When a reference is provided, the Monitor compares the current phase duration against the baseline from a previous "golden" ATPG run on a similar design. When creating the baseline, the Profiler extracted the duration of each phase and saved it in a CSV file. The Monitor loads this CSV at startup, giving it expected durations for all phases. During monitoring, after identifying the current phase and calculating how long it's been running, the Monitor looks up that phase in the baseline CSV to get the expected duration. It then applies a threshold multiplier (default 1.5×) — if the current duration is at least 1.5 times the baseline duration, it triggers a slow task alert. The 1.5× threshold was chosen to account for normal run-to-run variation while still catching genuinely slow executions.

When no reference is provided, the Monitor uses an absolute maximum duration threshold that is configured in the system settings (default 30 hours). This threshold applies uniformly to all tasks regardless of which specific phase is running. If any task has been executing for 30 hours or more, the Monitor triggers a slow task alert. The alert message is formatted differently to indicate that an absolute threshold was exceeded rather than a relative comparison: *"task_name is taking too long: 31:15:42 (max: 30:00:00)"* versus the baseline comparison message *"task_name is taking too much time comparing to the reference."*.

Using 30 hours as default is based on empirical observation of typical ATPG runs in the target environment. Most individual tasks complete within 12 hours, so 30 hours represents a genuinely excessive duration that likely indicates a problem. The threshold is configurable through the JSON configuration file, allowing adjustment for environments with different performance characteristics.

To avoid false alarms for very short phases, there is a minimum duration threshold of 8 minutes regardless of whether a reference is provided. If a phase has been running for less than 8 minutes, it won't trigger a slow alert even if it exceeds the threshold (whether relative or absolute). This prevents alerts for tiny phases where small absolute differences might exceed

percentage thresholds or where brief delays don't matter.

5.5.3 Detecting low coverage

This capability checks whether test coverage during the main pattern generation phase execution meets expectations, providing early warning of quality problems.

The detection works by waiting for this phase to complete (detected by finding its `task_break` marker in the log), then scanning backward through recent log lines to find the last coverage report that appeared before the phase ended. This coverage value represents the final coverage achieved during this important phase.

When a reference is provided, the baseline comes from the reference CSV file that contains coverage evolution data from the golden run. The Monitor reads this CSV and takes the maximum value of test coverage. The tool uses a fixed comparison threshold of 2 percentage points. If the current phase test coverage is at least 2 percentage points below the baseline maximum (for example, baseline is 88.5% but current is only 85.0%), the Monitor triggers a low coverage alert.

When no reference is provided, the Monitor uses an absolute minimum coverage threshold configured in the system settings (default 95%). If the main phase final coverage is below this minimum, the Monitor triggers a low coverage alert. The alert message clearly indicates this is an absolute threshold check: *"main_phase test coverage is too low: 92.50% (minimum required: 95.00%)"* versus the baseline comparison message that shows the delta: *"main_phase test coverage is too low: 85.00% (reference: 88.50%, delta: -3.50%)"*.

The 95% default minimum was chosen based on typical quality requirements in production ATPG workflows. Most designs target 95% or higher final coverage, so main phase coverage significantly below this indicates the run will likely fail to meet final targets. The threshold is configurable through the JSON configuration file, allowing adjustment for designs with different coverage requirements.

The coverage alert is fired only once per monitoring session. After sending the alert, the Monitor sets a flag that prevents duplicate alerts for the same issue. This flag persists for the entire monitoring session regardless of how many more polls occur.

5.5.4 Email alert system

All Monitor alerts are delivered via email. The alert email uses HTML formatting to present information clearly. It includes a table with rows for the alert message, log file path, current timestamp, bootup timestamp (when the ATPG run started), current phase name, phase duration, and any error messages. For stuck alerts specifically, the duration field shows how long the log has been unchanged rather than the phase duration.

The HTML formatting makes the alert easy to read on any device.

5.6 Updating references

Both the Profiler and Monitor need reference baselines to perform comparisons, so reference management has been integrated as a shared feature accessible from both modes. This design makes sense because creating a baseline is fundamentally a specialized profiling operation that extracts and exports specific data from a completed ATPG run.

A baseline represents a ”golden” ATPG run that exemplifies expected behavior. Engineers create baselines from successful runs that meet all quality criteria and perform well. Once created, this baseline serves as a comparison point for all future runs on the same or similar designs.

When using the Monitor, selecting a reference by name (like `design_A`) automatically provides the Monitor with access to the CSV files required for its detection algorithms: the task duration CSV for detecting slow tasks, and the coverage time-series CSV for detecting low coverage. The Monitor derives the coverage CSV path automatically by replacing `_task_duration.csv` with `_coverage_vs_time.csv` in the task duration path. This design means users only need to select one reference name, and the system handles locating all necessary data files.

There are several situations where creating a new baseline makes sense. When starting work on a new design, the first successful ATPG run becomes the initial baseline. When ATPG parameters are tuned for better performance, the improved run becomes a new baseline. When the design undergoes significant changes, a fresh baseline should be created to reflect the new expected behavior.

The tool encourages baseline creation by integrating it into the normal workflow. When running the Profiler, after selecting the log file to analyze, the user is prompted whether to compare against an existing baseline or create a new one. This prompt makes baseline creation a natural step rather than a separate complicated process.

When creating a baseline, the system generates several CSV files that capture different aspects of the golden run. These CSV files are simple text files that can be inspected manually if needed, but they are primarily intended to be read by the tool.

Task Duration CSV (`Design_A_task_duration.csv`): Contains task names and their elapsed times in seconds. The Monitor uses this file to detect when tasks are running significantly longer than expected.

Coverage Time-Series CSV (`Design_A_coverage_vs_time.csv`): Contains time-series data showing how coverage evolved during pattern generation, with columns for elapsed time, test coverage percentage, and cumulative pattern count. The Profiler uses this for generating comparison plots, and the Monitor uses it to establish coverage baselines for the main pattern generation phase.

Metadata CSV (`Design_A_metadata.csv`): Contains run metadata including bootup time, run ID, completion status, and error messages if any. This information provides context for identifying and tracking specific ATPG runs.

Create Patterns Options CSV (`Design_A_cp_options.csv`): Stores the options used in different `create_patterns` commands throughout the run, including execution mode, coverage effort level, and compaction settings. These parameters directly affect pattern generation behavior and final results.

Multiprocessing CSV (`Design_A_multiprocessing.csv`): Contains multiprocessing configuration including thread count, grid scheduling delays, and optimization settings. These settings impact run performance and parallelization efficiency.

All these files are stored in a directory named after the baseline. For example, creating a baseline named `Design_A` creates a directory `reference_logs/Design_A/` containing the five CSV files listed above.

CSV format was chosen because these files are human-readable plain text that can be opened in any text editor or spreadsheet application. If something goes wrong or if an engineer wants to understand what's in the baseline, they can simply open the CSV file and look. This transparency is valuable for debugging and building user trust.

Before saving a baseline, the system validates the source log file. It checks that the log contains the completion marker, verifying this was a successful run that finished normally. It also performs sanity checks on extracted metrics: coverage percentages should be between 0 and 100, pattern counts should be positive integers, and phase durations should sum to approximately the total runtime. These validations prevent creation of corrupted baselines from bad log files.

```
Select a golden reference (or skip):
  [0] Skip
  [1] Design_A
  [2] Design_B
  [3] Add new golden reference from a log file

Your choice: 3
Enter the path to the completed golden log file: /log/path/to/use/as/reference.log
Enter a name for this golden reference (e.g. ref1): Design_A

 Generating reference 'Design_A' from: /log/path/to/use/as/reference.log
======================== RUNNING LOG ANALYSIS ========================

Analysing: /log/path/to/use/as/reference.log ...

High level report: completed.

Medium-level report: completed.

  Reference metrics written to: reference_logs/Design_A/Design_A_metrics.csv
  Metadata written to: reference_logs/Design_A/Design_A_metadata.csv
  CP options written to: reference_logs/Design_A/Design_A_cp_options.csv
  Multiprocessing metrics written to: reference_logs/Design_A/Design_A_multiprocessing.csv
======================== LOG ANALYSIS COMPLETED ========================

Do you want to save this reference for future use? [y/N]: y
```

Figure 5.10: Reference Management Interface

5.7 Configuration file

The behavior of the tool is partly controlled by a JSON configuration file stored in the project root. This file contains the most commonly used paths and default values, so that the user does not need to enter them manually every time.

In particular, the configuration file stores:

- reference dataset paths, used in the comparison and monitoring modes
- default output names, used when generating reports

The file is read at startup and updated automatically when a new reference is created. It can also be edited manually if the user wants to add or remove entries. This makes the tool more flexible and easier to adapt to different working environments.

5.8 Summary

The ATPG Run Analyzer provides a structured and easy-to-use interface for working with ATPG log files. Its design is centered around three main workflows: analyzing completed runs, monitoring active runs, and managing reference data. By combining these functions into a single command-line application, the tool makes it easier to follow ATPG runs from start to finish.

At the same time, the tool is designed with reliability in mind. The use of automated tests at startup, together with a simple menu-driven interface and reusable configuration data, helps reduce errors and improves the user experience. In this way, the ATPG Run Analyzer acts as a bridge between raw log data and practical engineering analysis.

Chapter 6

Results and Tool Validation

This chapter presents the results obtained from deploying the ATPG Run Analyzer in a real industrial DFT workflow. The tool has been tested on real ATPG log files from production designs, covering different scenarios across both Profiler and Monitor mode.

It is worth noting that some of the values shown in the following results, such as log file paths, exact fault counts, coverage numbers, and phase names, have been modified to avoid exposing sensitive company information. However, the structure and the relative relationships between the values have been preserved, so the results still accurately represent the behaviour and the capabilities of the tool in a real environment. The conclusions that can be drawn from the data remain valid.

6.1 Profiler mode results

6.1.1 Analysis of a complete ATPG run

This section presents results obtained from the analysis of a complete ATPG run without providing a reference baseline. Figure 6.1 shows the user interaction with the Profiler mode through the command-line interface.

```
What do you want to run?
  [1] Profiler
  [2] Monitor

Your choice: 1

Enter the full log file path: /log/path/to/analyze.log

Enter output name (or press Enter for default 'Profiler/tool_excel_reports/260726_0900_Profiling.xlsx'): Example

Enter minimum test coverage target % for report (or press Enter for 96): 95

Enter recipient username for email report (or press Enter for 'current_user'):

 Log file   : /log/path/to/analyze.log
 Output name: Profiler/tool_excel_reports/260726_0900_Example.xlsx
 Report to  : current_user@email_domain.com

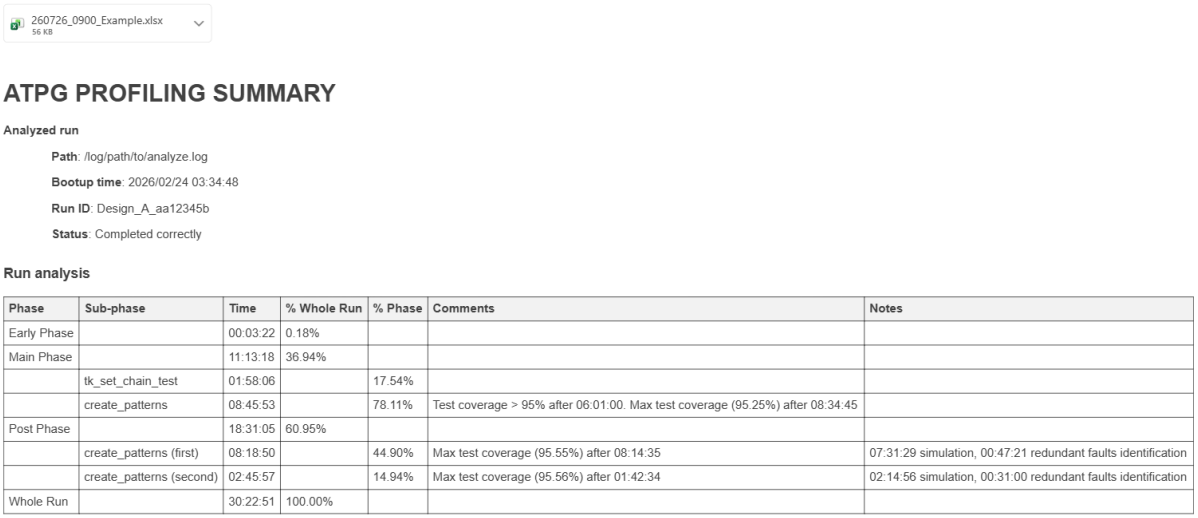

Select a golden reference (or skip):
  [0] Skip
  [1] Design_A
  [2] Design_B
  [3] Add new golden reference from a log file

Your choice: 0
```

Figure 6.1: Profiler mode user input sequence for a complete run analysis

The user provided the log path `/log/path/to/analyze.log`, specified the output name as `Example`, set a target test coverage of 95%, chose to use their own username for email notification, and selected to skip the reference baseline selection. The tool then proceeded to parse the log file and generate both the Excel report and the HTML email summary.

Figure 6.2 shows the HTML email summary generated by the tool. The summary provides a comprehensive overview of the ATPG run at a glance.



260726_0900_Example.xlsx
56 KB

ATPG PROFILING SUMMARY

Analyzed run

Path: /log/path/to/analyze.log

Bootup time: 2026/02/24 03:34:48

Run ID: Design_A_aa12345b

Status: Completed correctly

Run analysis

Phase	Sub-phase	Time	% Whole Run	% Phase	Comments	Notes
Early Phase		00:03:22	0.18%			
Main Phase		11:13:18	36.94%			
	tk_set_chain_test	01:58:06		17.54%		
	create_patterns	08:45:53		78.11%	Test coverage > 95% after 06:01:00. Max test coverage (95.25%) after 08:34:45	
Post Phase		18:31:05	60.95%			
	create_patterns (first)	08:18:50		44.90%	Max test coverage (95.55%) after 08:14:35	07:31:29 simulation, 00:47:21 redundant faults identification
	create_patterns (second)	02:45:57		14.94%	Max test coverage (95.56%) after 01:42:34	02:14:56 simulation, 00:31:00 redundant faults identification
Whole Run		30:22:51	100.00%			

Figure 6.2: HTML email summary for completed ATPG run

The analyzed run started on 2026/02/24 at 03:34:48 with Run ID `Design_A_aa12345b` and completed successfully. The run analysis shows a clear breakdown across three major phases:

- **Early Phase**: 00:03:22 (0.18% of total run time)

- **Main Phase**: 11:13:18 (36.94% of total run time)
- **Post Phase**: 18:31:05 (60.95% of total run time)

The total run duration was 30:22:51. Within the Main Phase, the `tk_set_chain_test` command took 01:58:06 (17.54% of the phase), while the `create_patterns` command consumed 08:45:53 (78.11% of the phase). The tool reports that test coverage exceeded 95% after 06:01:00, reaching a maximum of 95.25% after 08:34:45.

The Post Phase included two sequential `create_patterns` invocations. The first took 08:18:50 (44.90% of the phase) and achieved 95.55% maximum test coverage after 08:14:35, requiring 07:31:29 for simulation and 00:47:21 for redundant fault identification. The second invocation consumed 02:45:57 (14.94% of the phase) and achieved 95.56% maximum test coverage after 01:42:34, with 02:14:56 for simulation and 00:31:00 for redundant fault identification.

The Excel report generated by the Profiler contains multiple worksheets that provide detailed analysis. Figure 6.3 shows an excerpt from the phase breakdown sheet.

LINE	TASK	MONTH	DAY	HOUR	MINUTE	SECOND	TIMESTAMP (s)	ABSOLUTE TIMESTAMP (s)	TIME NEEDED (s)	PERCENTAGE OF TIME
848	Bootup_start	2	24	3	34	48	12888	0	0	0
1006	Bootup_end	2	24	3	34	53	12893	5	5	0
1502	Task_break:read_design	2	24	3	36	39	12999	111	106	0.1
1504	Task_break:pre_set_current_design	2	24	3	36	39	12999	111	0	0
1547	Task_break:set_current_design	2	24	3	38	31	13111	223	112	0.1
23147	Task_break:read_pdl	2	24	3	38	35	13115	227	4	0

Figure 6.3: Phase breakdown with timestamps and duration calculations

This sheet displays each task with its precise timestamp broken down into individual components (month, day, hour, minute, second), the absolute timestamp from run start, and the time needed to complete each task. The rightmost column shows the percentage of total runtime consumed by each task, with color coding to immediately highlight the most time-consuming operations. Tasks consuming more than 10% of total runtime are highlighted in orange, while shorter tasks appear in green.

Figure 6.4 shows another section of the same phase breakdown sheet, focusing on the most time-consuming tasks.

34244	Task_break:pre_earlyphase	2	24	4	9	34	14974	2086	1	0
34447	Task_break:earlyphase	2	24	4	12	56	15176	2288	202	0.18
34452	Task_break:pre_mainphase	2	24	4	12	56	15176	2288	0	0
89579	Task_break:mainphase	2	24	15	26	14	55574	42686	40398	36.94
123863	Task_break:postphase	2	25	9	57	19	122239	109351	66665	60.95

Figure 6.4: Phase breakdown sheet showing the most time-consuming tasks

This view continues from the same sheet, displaying the critical task boundaries such as `Task_break:earlyphase`, `Task_break:mainphase`, and `Task_break:postphase`. The color-coded percentage column allows engineers to immediately identify which phases consumed the most time, with the Main Phase at 36.94% and Post Phase at 60.95% clearly highlighted in orange.

Figure 6.5 presents the medium-level analysis sheet for the Main Phase.

34557	pattern_creation	2	24	4	13	20	15200	24	0	0
34558	tk_create_patterns	2	24	4	13	20	15200	24	0	0
37002	pattern_writing	2	24	12	59	13	46753	31577	31553	78.11
37003	tk_set_chain_test	2	24	12	59	13	46753	31577	0	0
37014	tk_write_tsdb_data	2	24	14	57	19	53839	38663	7086	17.54

Figure 6.5: Medium-level sheet showing command timing within the Main Phase

This sheet provides a breakdown of individual commands within the Main Phase. Each line shows the duration and percentage relative to the Main Phase total duration. For example, the `create_patterns` command consumed 78.11% of the Main Phase time, while the `tk_set_chain_test` command took 17.54% of the Main Phase duration. Other operations such as `pattern_writing` show their individual time contributions as percentages of the Main Phase.

Figure 6.6 displays the pattern generation iteration details.

BLOCK	ITERATION	SIMULATED PATTERNS	DETECTED FAULTS	TEST COVERAGE (%)	CPU TIME (s)	TASK TIME (s)
1	1	64	10000000	45	812.99	328.27
1	2	128	2000000	52.5	1114.11	301.12
1	3	192	1200000	56.5	1638.44	524.33
1	4	256	800000	60	2678.09	1039.65

Figure 6.6: Per-iteration pattern generation metrics

This table, an excerpt of the `create_patterns` analysis sheet, shows the progression of pattern generation across multiple iterations. Each row represents one iteration, displaying the block number, iteration number within that block, number of simulated patterns, detected faults, test coverage percentage, CPU time consumed, and task time. This detailed view allows engineers to understand how coverage improved over time and identify any iterations that consumed unusually high computational resources.

6.1.2 Comparative analysis with reference baseline

When a reference baseline is provided, the Profiler generates an extended report that includes side-by-side comparison against the golden run. Figure 6.7 shows the comparative email summary.



Figure 6.7: Email summary with reference comparison - analyzed and reference run information

The summary clearly identifies both the analyzed run (path `/log/path/to/analyze_and_compare.log`, bootup time 2026/02/14 07:51:22) and the reference run (path `/log/path/to/reference_Design_A_run.log`, bootup time 2026/01/13 09:35:13). Both runs completed correctly, establishing a solid foundation for comparison. The standard profiling table for the analyzed run is still present.

Figure 6.8 presents the comparative metrics table.

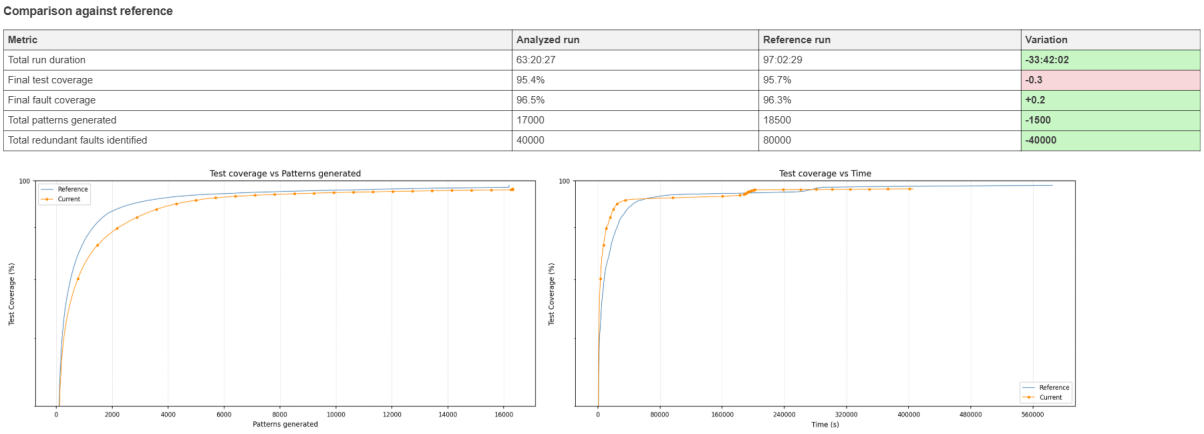


Figure 6.8: Comparison table showing metrics delta between current and reference runs

The comparison table displays key metrics side by side. The analyzed run shows a total duration of 63:20:27 compared to the reference duration of 97:02:29, representing a decrease. The final test coverage achieved is 95.4% versus the reference 95.7%, showing a slight decrease highlighted in red. The total patterns generated (17000) is lower than the reference (18500), and the tool detected 40000 redundant faults identified versus 80000 in the reference. Variations that are considered an improvement have a green cell background, red otherwise.

Below the table, two embedded charts visualize test coverage progression. The left chart shows ”Test coverage vs Patterns generated” comparing the current run (orange line) against the reference (blue line). The right chart displays ”Test coverage vs Time” with the same color coding, allowing engineers to see at a glance whether the current run converged to target coverage faster or slower than the baseline.

Figure 6.9 shows additional comparative visualizations embedded in the email: the options used for `create_patterns` commands and multiprocessing options. This information is reported since it affects the metrics of interest. Note that the specific option names and values shown in the tables have been redacted as they contain proprietary configuration information.

ATPG Options Comparison

Options	Analyzed run	Reference run
Main Phase	-set_abort_limit $low_abort_limit -auto -coverage_effort $cov_effort_value -compact $compaction_level	-set_abort_limit $low_abort_limit -auto -coverage_effort $cov_effort_value -compact $compaction_level
Post Phase (first)	-set_abort_limit $low_abort_limit -auto -coverage_effort $cov_effort_value -compact $compaction_level	-set_abort_limit $high_abort_limit -auto -coverage_effort $cov_effort_value -compact $compaction_level
Post Phase (second)	-set_abort_limit $high_abort_limit -auto -coverage_effort $cov_effort_value -compact $compaction_level	-set_abort_limit $low_abort_limit -auto -coverage_effort $cov_effort_value -compact $compaction_level
Post Phase (third)	-	-set_abort_limit $low_abort_limit -auto -coverage_effort $cov_effort_value -compact $compaction_level

Multiprocessing Options Comparison

Option	Analyzed run	Reference run
Total Threads	$num_threads	$num_threads
Grid Delay	$grid_delay_in_minutes	0
-optimize_secondaries	-	$optimize_secondaries_option

Figure 6.9: Additional comparison visualizations in email summary

An additional Excel file is attached to the email. The sheet shown in Figure 6.10 includes a tabular and graphical visualization of the metrics used in the summary email as well as patterns generated and test coverage per phase.

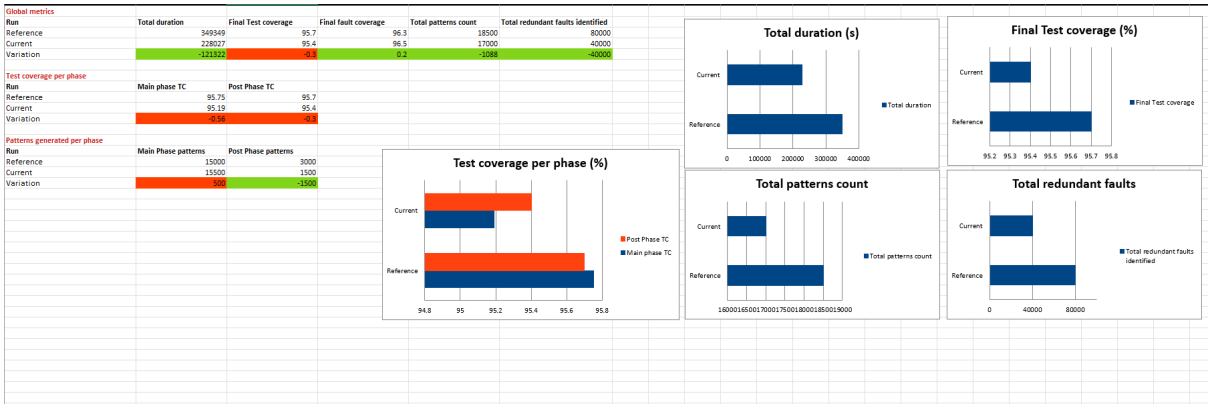


Figure 6.10: Excel comparison charts for multiple metrics

Figure 6.11 shows the second sheet of the new Excel file where the plots of Test Coverage vs. Time and Test Coverage vs. Patterns generated are displayed.

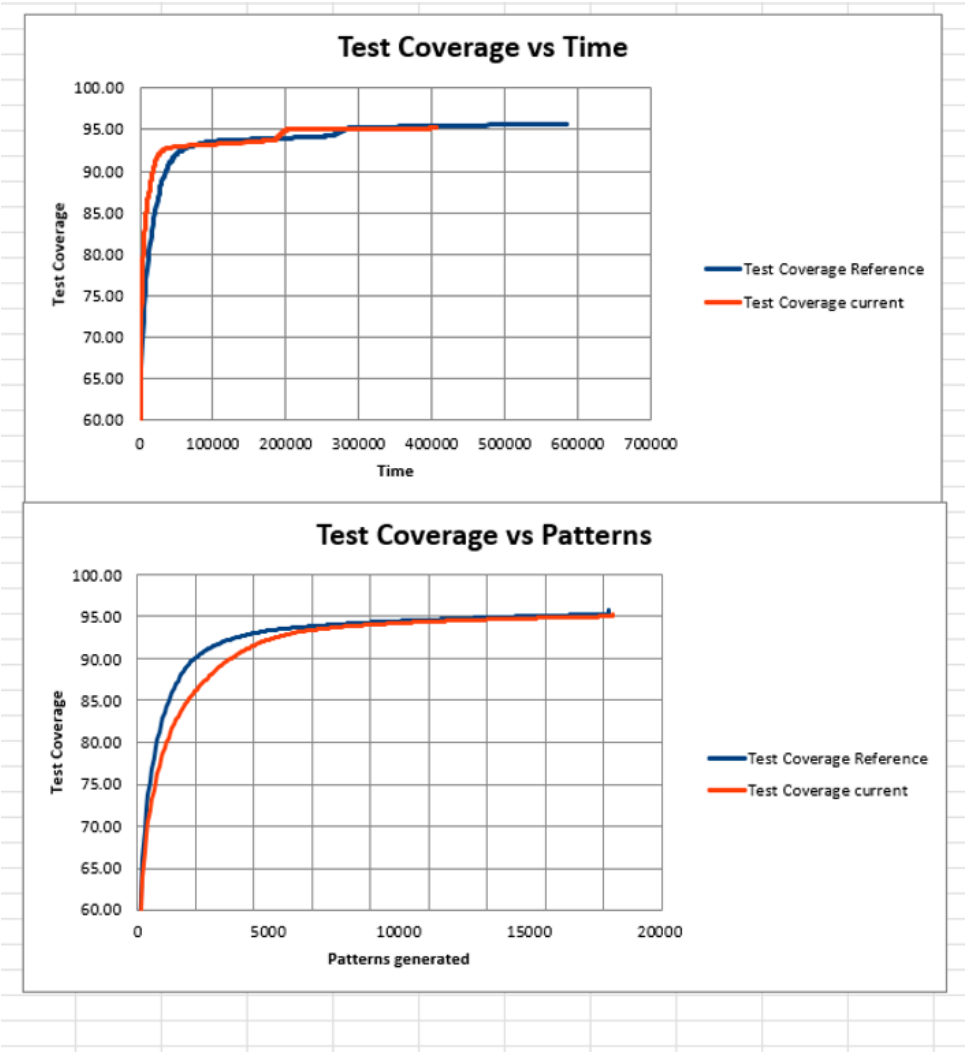


Figure 6.11: Test Coverage vs. Time and Test Coverage vs. Patterns generated plots

This comparison sheet presents all metrics in a structured table format with three columns: metric name, analyzed run value, reference run value, and variation. Cells are color-coded based on performance: green indicates improvement over the reference, red indicates degradation. This allows engineers to quickly scan the table and identify areas requiring investigation. For example, if test coverage decreased or runtime increased significantly, those cells would appear in red, immediately drawing attention.

6.1.3 Incomplete run handling

The tool is designed to handle incomplete or failed ATPG runs gracefully. Figure 6.12 demonstrates the behavior when analyzing an incomplete log file.

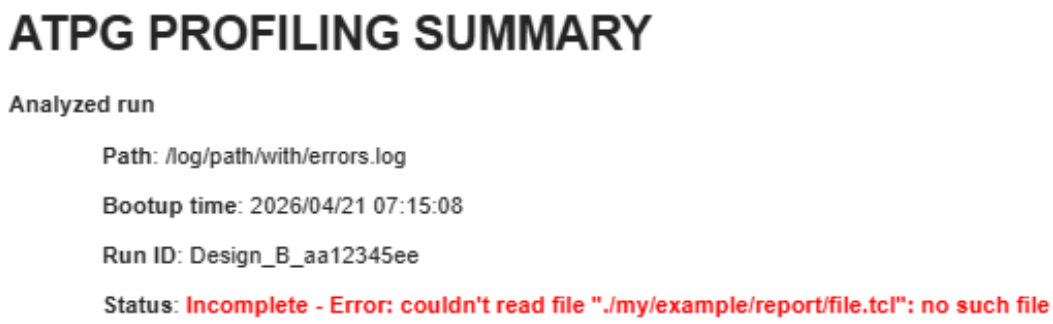


Figure 6.12: Email summary for an incomplete ATPG run with error reporting

When the tool detects that a log file does not contain the normal completion marker, it generates a warning status. The email summary shows the analyzed run path (`/log/path/with/errors.log`), bootup time (2026/04/21 07:15:08), and Run ID (`Design_B_aa12345ee`). Most importantly, the status field displays in red: **"Incomplete - Error: could not read file './my/example/report/file.tcl': no such file"**. This error message is extracted directly from the log file and included in the report, giving the engineer immediate context about what probably went wrong. In fact, this is the last error detected before the run ended without the `exit -f` line.

Despite the incomplete state, the tool still extracts all available data up to the point of failure and generates a partial report. This allows engineers to understand how far the run progressed, which phases completed successfully, and where exactly the failure occurred. This capability is particularly valuable when debugging ATPG script issues or investigating unexpected job terminations.

6.2 Monitor mode results

Monitor mode provides continuous oversight of ongoing ATPG runs by periodically polling the log file and checking for anomalies. This section presents examples of the three main alert types: stuck run detection, slow task detection, and low coverage alerts, demonstrating monitoring behavior both with and without reference baselines.

6.2.1 Monitoring with reference baseline

When a reference baseline is provided, the Monitor performs relative comparisons against the golden run to detect anomalies. The following figures show the email alerts generated when different types of issues are detected.

Figure 6.13 shows the slow task alert email received when the tool detects that the current task is taking significantly longer than expected compared to the reference baseline.

ATPG Monitor Alert

Message	mainphase is taking too much time comparing to the reference.
Log file	/log/path/to/monitor.log
Alert time	2026/07/27 16:32:15
Bootup time	2026/07/26 08:16:28
Current task	mainphase
Task duration / stuck for	06:15:46

Figure 6.13: Slow task alert email with reference comparison

In this example, the Main Phase has been running for 06:15:46, which exceeds the expected duration from the reference run. The alert message states ”mainphase is taking too much time comparing to the reference.” The email includes the log file path (`/log/path/to/monitor.log`), alert time (2026/07/27 16:32:15), bootup time (2026/07/26 08:16:28), current task (mainphase), and the task duration (06:15:46). This allows the engineer to assess whether the run is progressing abnormally slowly and may require investigation.

Figure 6.14 demonstrates the stuck run alert.

ATPG Monitor Alert

Message	Log file has not changed for 02:00:00.
Log file	/log/path/to/monitor.log
Alert time	2026/07/26 16:32:15
Bootup time	2026/07/26 08:15:42
Current task	earlyphase
Task duration / stuck for	02:00:00
Last error	Error: Licence Timeout

Figure 6.14: Stuck run alert email showing log file has stopped growing

The stuck alert is triggered when the log file has not changed for two consecutive polling intervals. In this example, the message states ”Log file has not changed for 02:00:00.” The run was in the Early Phase when it became stuck. Importantly, the email also reports the last error found in the log file: ”Error: Licence Timeout”. This provides immediate context that can help diagnose the cause of the stuck condition. The stuck detection mechanism is independent of whether a reference baseline is provided, as it simply monitors whether the log continues to grow.

Figure 6.15 shows the low coverage alert.

ATPG Monitor Alert

Message	mainphase test coverage is too low: 82.45% (reference: 94.78%, delta: -12.33%)
Log file	/log/path/to/monitor.log
Alert time	2026/07/26 16:32:15
Bootup time	2026/07/25 08:15:42
Current task	postphase
Task duration / stuck for	N/A

Figure 6.15: Low coverage alert email with delta from reference

This alert is generated when the Main Phase achieves a maximum test coverage that is significantly below the reference baseline. The message states ”mainphase test coverage is too low: 82.45% (reference: 94.78%, delta: -12.33%)”. The email shows that the run has progressed to the Post Phase, but the Main Phase failed to achieve adequate coverage. The delta calculation makes it immediately clear how much the coverage has degraded compared to the expected performance. This early warning allows engineers to investigate coverage issues or potentially abort the run before it completes.

6.2.2 Monitoring without reference baseline

When no reference baseline is provided, the Monitor uses absolute thresholds configured in the system. The alert messages are adapted to reflect this different detection mode.

Figure 6.16 shows the slow task alert without a reference.

ATPG Monitor Alert

Message	mainphase is taking too long: 32:15:47 (max: 30:00:00).
Log file	/log/path/to/monitor.log
Alert time	2026/07/27 16:32:15
Bootup time	2026/07/26 08:16:28
Current task	mainphase
Task duration / stuck for	32:15:47

Figure 6.16: Slow task alert email using absolute threshold

Without a reference, the alert message changes to ”mainphase is taking too long: 32:15:47 (max: 30:00:00).” Instead of comparing against a baseline duration, the tool compares against the configured maximum duration threshold (30 hours by default). This mode is useful when monitoring new designs or when ATPG parameters have changed significantly, making old baselines invalid.

Figure 6.17 demonstrates the low coverage alert without reference.

ATPG Monitor Alert

Message	mainphase test coverage is too low: 88.23% (minimum required: 95.00%)
Log file	/log/path/to/monitor.log
Alert time	2026/07/27 16:32:15
Bootup time	2026/07/26 08:16:28
Current task	postphase
Task duration / stuck for	N/A

Figure 6.17: Low coverage alert email using absolute minimum threshold

The message states ”mainphase test coverage is too low: 88.23% (minimum required: 95.00%)”. Instead of reporting a delta from a reference value, the alert compares against the configured minimum coverage threshold (95% by default). The stuck alert behavior remains identical regardless of whether a reference is provided, as it depends only on whether the log file continues to grow between polling intervals.

6.3 Time savings and efficiency gains

The ATPG Run Analyzer saves a lot of time compared to analyzing log files manually. In the worst case scenario, when using Profiler mode with reference comparison and adding a new reference baseline, the tool finishes everything and generates all reports in about 2 minutes. If an engineer had to do the same work manually, extracting timestamps, calculating how long each phase took, parsing coverage data, finding problems, and writing a complete report—it would take more than 3 hours. Since engineers typically need to do this kind of detailed ATPG analysis 10 to 15 times per year, the tool saves up to 2 working weeks per engineer every year.

For Monitor mode, it’s harder to measure the time savings directly, but the real value is in preventing wasted time rather than just speeding up a task. Without automated monitoring, engineers have to either check log files manually every hour or so during a long run, or wait until the end to find out something went wrong many hours ago. The Monitor removes the need for manual checking completely, so engineers can work on other things while knowing they’ll get an alert right away if something goes wrong. Even more importantly, catching problems early—like stuck runs, license timeouts, or low coverage, lets engineers stop bad jobs within minutes instead of hours, which saves expensive compute resources and reduces the time needed to fix things and try again. For example, if a slow task or coverage problem is detected 6 hours into a 30-hour run, the engineer can investigate and restart with better settings, avoiding 24 more hours of wasted computation. Over multiple ATPG projects, this early warning capability can save several days of compute time and speed up the whole project timeline.

Chapter 7

Conclusions

This thesis presented the ATPG Run Analyzer, a tool developed to automate the analysis and monitoring of Siemens Tessent ATPG log files in an industrial environment. The work started from a real problem that engineers face every day: ATPG runs can take many hours and produce log files with tens of thousands of lines, making manual inspection slow, tedious, and prone to mistakes. When engineers are working on multiple designs at the same time, keeping track of everything becomes almost impossible without some kind of automation. On top of that, there was no easy way to detect problems while a run is still in progress, so issues like stuck processes or tasks taking too long often went unnoticed until the job finished or crashed, wasting hours of expensive compute time.

To provide the necessary background, the thesis first explained the fundamentals of Design for Test and why it matters in the semiconductor industry. It covered how manufacturing defects happen in chips and why test patterns are needed to catch them. The discussion included the main fault models used in practice, like stuck-at faults and transition faults, and explained how scan-based testing and ATPG work together to achieve high test coverage. Understanding these concepts made it clear why metrics like test coverage, pattern count, and fault detection are so important when evaluating an ATPG run. The thesis then moved to the industrial side of things, describing how tools like Siemens Tessent are actually used in production environments. It explained the typical ATPG workflow with its different phases: early setup, main pattern generation, post-processing and what the log files look like. Understanding this structure was key to designing a tool that could reliably parse these logs and pull out the important information.

Since the tool needed to work reliably in a production setting, a lot of attention went into software engineering practices, especially Test-Driven Development. A comprehensive test suite was built with 69 unit tests organized into four modules that cover time calculations, log parsing, metric extraction, and email generation. The tool runs this entire test suite automatically every time it starts up, so any problems in the environment or bugs in the code get caught right away before any real data is processed. This gives confidence that the tool won't silently give wrong results, which is really important when those results are used to make decisions about design quality.

The tool itself has a modular design with two main operating modes. Profiler mode is for completed runs and does a deep dive into the entire log file. It pulls out phase durations, pattern generation metrics, coverage progression, and fault statistics, then creates multi-sheet Excel reports with charts and sends HTML email summaries to engineers. When a reference baseline from a previous good run is available, Profiler does a side-by-side comparison, showing differences in timing, coverage, and pattern counts with color-coded tables and charts. This comparison feature is really useful for spotting performance problems and understanding why a new run behaved differently than expected. Monitor mode works differently: it's designed for runs that are still going. It keeps checking the log file at regular intervals, looking for three types of problems: stuck runs where the log file stops updating, slow tasks that are taking longer than they should, and low coverage when test quality isn't meeting targets. Monitor can work with or without a reference baseline, comparing against historical data when it's available or using fixed thresholds for new designs.

The tool was deployed in the real production environment and tested on actual log files from different scenarios. The results showed that Profiler successfully extracts all the relevant metrics and presents them in a clear format that lets engineers spot bottlenecks and coverage problems immediately. The comparison feature turned out to be particularly valuable since it shows at a glance whether a new run did better or worse than a baseline. Monitor mode effectively caught stuck runs, slow tasks, and coverage issues in real time, with email alerts that let engineers fix things quickly. In terms of time savings, the analysis showed that the tool finishes a complete profiling with comparison in about 2 minutes, while doing the same work manually would take over 3 hours. Since these analyses are typically needed 10 to 15 times per year, that adds up to saving almost 2 working weeks per engineer every year. The value of Monitor mode is harder to measure directly, but it prevents wasted time and compute resources. For example, catching a problem 6 hours into a 30-hour run means the engineer can investigate and restart with better settings, avoiding 24 more hours of wasted computation.

The tool successfully meets all its design goals and is actively being used in production, but there are several ways it could be improved in the future to provide even more value:

- **Centralized Dashboard**: Right now the tool sends Excel reports and HTML emails to individual engineers, but there's no central place where all results are stored and everyone on the team can access them. A web-based dashboard would solve this by automatically, or after user's approval, uploading all reports to a shared database. The dashboard could include a searchable archive of all past runs organized by design, date, and engineer, interactive charts showing trends in coverage and runtime over time, side-by-side comparison views for any two runs, and filtering tools to quickly find runs that meet certain criteria, like all runs that missed coverage targets or took too long. This would turn the tool from a personal utility into a team knowledge base, making it easier to learn from past runs, share findings with colleagues, and spot long-term patterns that aren't obvious when looking at individual runs.

- **Integration with Job Scheduling Systems**: In most industrial settings, ATPG runs are launched through job schedulers like LSF or Grid Engine. Right now, users have to manually tell the tool which log file to monitor. A better approach would be to integrate directly with the job scheduler so the tool automatically detects when a new ATPG job starts and begins monitoring it without any manual setup. Similarly, when a job finishes, Profiler could run automatically and generate the report without anyone having to remember to launch it. This would make the tool almost invisible to users: runs would be monitored and analyzed in the background, and engineers would just get alerts and reports when needed. This would be especially useful when dozens of ATPG runs are happening every day across different designs, since it would guarantee that every run gets monitored consistently.

- **Machine Learning-Based Anomaly Detection**: The current Monitor mode uses fixed thresholds and comparisons to reference runs to detect problems. This works well but requires engineers to manually configure what counts as abnormal for each design. A smarter approach would use machine learning to automatically learn what's normal based on historical data. A model trained on successful runs could learn the typical duration of each phase, the expected coverage curve, and normal pattern generation efficiency for a given design. When monitoring a new run, the model could predict what should be happening at each point and flag anything unusual without needing manual thresholds. This would make the tool more adaptive, especially for new designs that don't have reference baselines yet, and would save engineers from having to configure thresholds manually.

One of the tool's biggest strengths that makes these future improvements possible is its comprehensive unit test suite. The 69 tests that cover all critical functionality don't just ensure reliability right now but they also make the code easy to adapt when things change in the future. If the Tessent log format changes in a new software version, or if the team's workflow evolves to include new phases or different metrics, the unit tests act as a safety net that lets developers modify the code without worrying about breaking things. For example, if a new coverage metric needs to be extracted, a developer can write new parsing code and add tests to check that it works correctly, while all the existing tests keep running to make sure nothing else broke. This makes the tool maintainable over time and keeps the cost of future improvements low. The Test-Driven Development approach used in this project isn't just about quality assurance but it's an investment in making the tool adaptable and extensible for the long term.

The ATPG Run Analyzer shows how a focused automation tool can make a real difference in an industrial workflow. By solving specific problems that engineers deal with every day: manual log inspection, lack of visibility into running jobs, and difficulty comparing runs. The tool saves time on routine tasks and lets engineers focus on more valuable work like improving test quality and tuning ATPG settings. The project also demonstrates why good software engineering practices matter when building tools for production use. Using comprehensive

testing, modular design, and clear separation between components made development easier and resulted in a tool that's reliable, easy to maintain, and ready for future enhancements. The future improvements proposed here like centralized dashboards, automatic job integration and machine learning detection would add even more value and could transform it from a team-specific tool into a more general platform for ATPG run management and analysis. The solid foundation built through comprehensive testing and modular design makes these extensions realistic and ensures the tool can keep evolving as the team's needs change.

Bibliography

[1] Intel. “Billions of transistors make up a processor. a new video shows how they are made,” Intel Newsroom, Accessed: Jul. 27, 2026. [Online]. Available: https://newsroom.intel.com

[2] AnySilicon. “Overview and dynamics of scan testing,” Accessed: Jul. 27, 2026. [Online]. Available: https://anysilicon.com/overview-and-dynamics-of-scan-testing/

[3] *Tessent® scan and atpg user’s manual*, Siemens EDA, 2020.

[4] *Tessent® testkompress® user’s manual*, Siemens EDA, 2020.

[5] M. L. Bushnell and V. D. Agrawal, “Digital dft and scan design,” in *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, ser. Frontiers in Electronic Testing, vol. 17, Boston, MA: Springer, 2002, pp. 465–488, ISBN: 0-7923-7991-8. DOI: 10.1007/0-306-47040-3_14

[6] Python Software Foundation. “Unittest — unit testing framework,” Accessed: Jul. 27, 2026. [Online]. Available: https://docs.python.org/3/library/unittest.html#unittest.TestCase

[7] Python Software Foundation. “Unittest.mock — mock object library,” Accessed: Jul. 27, 2026. [Online]. Available: https://docs.python.org/3/library/unittest.mock.html#magic-mock

[8] IBM. “What is test-driven development (tdd)?” Accessed: Jul. 27, 2026. [Online]. Available: https://www.ibm.com/think/topics/test-driven-development

[9] pandas development team. “Pandas — python data analysis library,” Accessed: Jul. 27, 2026. [Online]. Available: https://pandas.pydata.org/docs/

[10] openpyxl contributors. “Openpyxl — a python library to read/write excel 2010 xlsx/xlsm files,” Accessed: Jul. 27, 2026. [Online]. Available: https://openpyxl.readthedocs.io/

[11] Matplotlib development team. “Matplotlib — visualization with python,” Accessed: Jul. 27, 2026. [Online]. Available: https://matplotlib.org/stable/

[12] Python Software Foundation. “Smtplib — smtp protocol client,” Accessed: Jul. 27, 2026. [Online]. Available: https://docs.python.org/3/library/smtplib.html