

# AutoHIL: LLM-Based ECU Functional Test Generation through Domain Knowledge Augmentation

SICHEN GONG, Fudan University, China

QICAI CHEN, Fudan University, China

BIHUAN CHEN\*, Fudan University, China

WENZHUO ZHANG, Fudan University, China

YUKUN GAO, ANHUI GuarDrive Safety Technology, China

XIN PENG, Fudan University, China

System functional testing is a critical step before the production release of automotive electronic control units (ECUs), especially for safety-critical functions. It is typically performed on hardware-in-the-loop (HIL) benches, and engineers translate functional requirements in natural language into executable test scripts using bench-specific APIs (or test harness primitives (THPs)). Existing research either focuses on generating abstract test cases that lack execution capability, or relies on bench-specific solutions that cannot be generalized. Consequently, industrial practice remains labor-intensive, and is hindered by three main challenges, i.e., the high learning cost of THPs, the incompleteness of functional requirements due to missing tacit domain knowledge (e.g., power management dependencies or system wake-up sequences), and the limited automation.

To address these challenges, we propose AutoHIL, an LLM-based, domain-knowledge-augmented framework for requirement-driven ECU functional test generation on HIL benches. AutoHIL automatically constructs a THP knowledge base to understand bench operations, extracts tacit domain knowledge from AUTOSAR ECU source code and configurations to augment requirements, and generates test scripts for the augmented requirements based on the THP knowledge base. We evaluate AutoHIL on two industrial ECUs, i.e., an Airbag Control Unit (ACU) on an in-house bench and an Active Safety Domain Master (ASDM) on a dSPACE platform. AutoHIL achieves test script executability of 90.33% on ACU and 81.31% on ASDM. We also manually evaluate the functional equivalence between generated scripts and manual scripts, and AutoHIL reaches a functional correctness of 89.69% on ACU and 84.00% on ASDM. We discover 39 and 25 new functional defects in ACU and ASDM, which have been patched by technicians from the supplier.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: ECU functional testing, test generation, hardware-in-the-loop testing, large language models

## ACM Reference Format:

Sichen Gong, Qicai Chen, Bihuan Chen, Wenzhuo Zhang, Yukun Gao, and Xin Peng. 2026. AutoHIL: LLM-Based ECU Functional Test Generation through Domain Knowledge Augmentation. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA064 (October 2026), 22 pages. <https://doi.org/10.1145/3832155>

\*Bihuan Chen is the corresponding author.

Authors' Contact Information: [Sichen Gong](mailto:scgong24@m.fudan.edu.cn), College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, [scgong24@m.fudan.edu.cn](mailto:scgong24@m.fudan.edu.cn); [Qicai Chen](mailto:qcchen23@m.fudan.edu.cn), College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, [qcchen23@m.fudan.edu.cn](mailto:qcchen23@m.fudan.edu.cn); [Bihuan Chen](mailto:bhchen@fudan.edu.cn), College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, [bhchen@fudan.edu.cn](mailto:bhchen@fudan.edu.cn); [Wenzhuo Zhang](mailto:wzzhang24@m.fudan.edu.cn), College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, [wzzhang24@m.fudan.edu.cn](mailto:wzzhang24@m.fudan.edu.cn); [Yukun Gao](mailto:gaoyukun@mychery.com), ANHUI GuarDrive Safety Technology, Shanghai, China, [gaoyukun@mychery.com](mailto:gaoyukun@mychery.com); [Xin Peng](mailto:pengxin@fudan.edu.cn), College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, [pengxin@fudan.edu.cn](mailto:pengxin@fudan.edu.cn).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2994-970X/2026/10-ARTISSTA064

<https://doi.org/10.1145/3832155>

## 1 Introduction

Automotive software encompasses a massive amount of code and spans hundreds of electronic control units (ECUs) and sensor/actuator nodes, leading to a significant increase in integration complexity and the risk of software defects [33]. For safety-critical functions like braking, airbags, chassis control, and powertrain management, systemic failures can result in unacceptable hazards. Therefore, rigorous ECU testing and validation is essential to ensure the safety and reliability of ECUs [23]. In recent years, large language models (LLMs) have demonstrated exceptional performance in test automation [10], leading to their increasing adoption in ECU testing [9, 41].

To support quality assurance and traceability, original equipment manufacturers (OEMs) and Tier-1 suppliers typically organize development around a V-model process [23]. As shown in Fig. 1, the left side of the V-model progresses from vehicle requirements to system requirements and design, software design, and software unit design. In the early stages of the right side, unit testing and software integration testing, conducted in software-in-the-loop (SIL), model-in-the-loop (MIL) or processor-in-the-loop (PIL) environments, are well-supported by mature toolchains (e.g., Simulink Test [34]) and established research [18, 42]. Our work focuses on the ECU system testing stage, where the integrated ECU system is validated against system requirements.

In industrial practice, ECU system testing commonly operates on hardware-in-the-loop (HIL) benches, where the ECU under test is connected to real-time, closed-loop simulations of in-vehicle networks and sensor/actuator models. HIL testing enables repeatable validation without risking real vehicles, and reduces debugging and testing costs [35]. Each HIL bench exposes open APIs for writing HIL test scripts.

While amounts of researches are focused on system functional safety and robustness using fault injection [1, 5, 29], our work specifically targets ECU system functional testing, which validates whether the functionality of the ECU meets functional requirements. Existing effort in ECU system functional testing primarily concentrates on generating formal or semi-formal test cases (e.g., test flows/specifications, sequence diagrams) from requirements [3, 12, 20, 28, 36, 44, 47]. While these approaches significantly advance the automation of test design, they do not execute on HIL benches, thus often neglecting to address the hardware characteristics and interface constraints of HIL environments. At the same time, a few works actually perform execution verification on HIL benches, but they depend on specific HIL platforms, and only achieve partial automation [16, 17, 26, 43].

Consequently, ECU system functional testing on HIL benches still heavily relies on manual efforts in industrial practice. Requirement engineers describe test requirements in natural language, which often require iterative refinement and multi-source information fusion [42]. Test engineers then map test requirements to executable test script steps [25], align HIL bench APIs with signal semantics and comprehend tacit domain knowledge within the AUTOSAR stack and system interactions, which is labor-intensive and time-consuming. When test scripts fail, test engineers localize failures using logs and waveforms [2]. In summary, ECU system functional testing faces three main challenges: (i) the high cost of learning and aligning domain-specific bench APIs (referred to as *test harness primitives* (THPs) in this work), (ii) the incompleteness of functional requirements due to the lack of tacit domain knowledge, and (iii) the limited automation of the whole test script generation process.

To address these challenges, we propose AutoHIL, an LLM-based framework designed for the end-to-end automation of ECU system functional test generation on HIL benches through domain

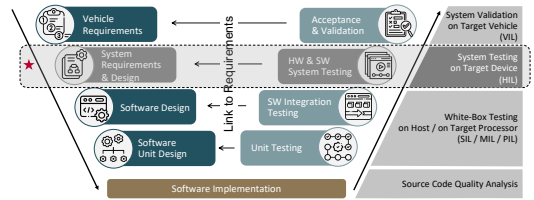


Fig. 1. V-model of ECU Development

knowledge augmentation. To achieve general adaption across different HIL benches, AUTOHIL automatically analyzes THPs of HIL benches to construct a THP knowledge base, allowing the LLM to retrieve and utilize bench-specific THPs via retrieval-augmented generation (RAG) in test script generation. To complement the tacit domain knowledge involved in the requirements, we extract domain concepts and functional logic from AUTOSAR ECU source code and configuration files to augment system functional requirements. Finally, to realize end-to-end test automation, we use the augmented requirements as input and the THP knowledge base as the retrieval source, and utilize LLM to decompose requirements into test steps aligned with THPs, format steps into behavior trees as an intermediate representation, and ultimately convert them into executable HIL test scripts.

Specifically, as AUTOHIL targets in-house testing, we assume that testers can access HIL bench interfaces (API documentation for commercial benches, or interface source code for in-house benches) and AUTOSAR C source code of the ECU under test. This assumption aligns with the industrial practice, where test engineers need HIL interfaces to write test scripts, and inspect source code of the ECU under test to clarify implementations and localize defects. AUTOHIL leverages this reality by turning scattered domain knowledge into structured information consumable by LLMs.

We evaluate AUTOHIL on two industrial ECUs from our industrial collaborator (a Tier-1 automotive supplier), i.e., an Airbag Control Unit (ACU) tested on an in-house HIL bench, and an Active Safety Domain Master (ASDM) on a commercial dSPACE platform. AUTOHIL successfully adapts to both bench environments, and generates executable test scripts for 90.33% and 81.31% of requirements on ACU and ASDM. Then, we assess the functional correctness of these scripts compared with manual scripts using manual sampling and evaluation. Specifically, 89.69% of ACU scripts and 84.00% of ASDM scripts are equivalent to manual scripts, demonstrating the correctness of our generated scripts. Moreover, AUTOHIL discovers 64 previously unknown functional defects (39 on ACU and 25 on ASDM) that were missed by manual testing. We also analyze the efficiency gain in the industrial workflow. AUTOHIL reduces the end-to-end effort from over a month to fewer than eight workdays, including script repair (a  $3.8\times$  to  $7.1\times$  speedup), with 74.21% of scripts requiring no manual edits. Finally, we conduct an ablation study to assess the impact of key components. The results confirm that the THP knowledge base is indispensable for enabling execution by preventing THP hallucinations, while requirement augmentation is critical for ensuring high functional correctness and defect detection capability; and the overall performance remains stable across different reasoning LLMs.

In summary, this work makes the following main contributions.

- We propose a THP knowledge base construction method for HIL benches, enabling the LLM to accurately understand and utilize bench-specific THPs.
- We propose a requirement augmentation method for AUTOSAR ECUs to refine the functional requirements into structured augmented requirements that incorporate tacit domain knowledge.
- We present AUTOHIL, the first end-to-end, LLM-based framework that automates ECU system functional test generation on HIL benches.
- We evaluate AUTOHIL on two industrial ECUs to demonstrate high executability and high functional correctness of our generated test scripts, and we also discover 64 new functional defects.

## 2 Background and Motivation

### 2.1 Test Harness Primitives in HIL Benches

In automotive industrial practice, HIL testing is widely adopted for system-level functional testing of ECUs. By placing the ECU within a real-time simulation environment on a bench, HIL provides reproducible and controllable capabilities for stimulation and observation [35, 37].

Both commercial HIL benches (e.g., dSPACE [14], Typhoon [22], Opal-RT [38], and NI [13]) and proprietary in-house HIL benches developed by OEMs or suppliers expose a set of operational primitives via programmable interfaces, allowing test scripts to manipulate the interaction between the bench and the ECU, typically including power management (power-up/down, ignition cycling), operating condition loading (loading specific road profiles or vehicle model parameters), bus communication (sending/receiving messages via CAN/LIN/Ethernet), signal injection and fault simulation (open circuit, short circuit, message loss), as well as real-time model control and data acquisition [1]. Commercial HIL platforms typically provide simulation kernels and hardware drivers in the form of closed binary, exposing only documented APIs and configuration interfaces without open-source code. Conversely, in-house HIL platforms generally implement a project-level encapsulation layer on top of underlying drivers, forming a library of callable interfaces.

We collectively refer to these interfaces, which can be directly invoked in test scripts to drive the interaction between the HIL bench and the ECU under test, as *test harness primitives* (THPs). Formally, each THP  $t_j$  can be viewed as a procedure that induces a transition in the environmental state, i.e.,  $t_j : \mathcal{E} \times \mathcal{P}_j \rightarrow \mathcal{E}'$ , where  $\mathcal{E}$  represents the current state of the environment constituted jointly by the HIL and the ECU under test,  $\mathcal{P}_j$  is the set of parameters for the interface, and  $\mathcal{E}'$  is the new state after the invocation. For the task of test generation, we do not manipulate  $\mathcal{E}$  directly; instead, THPs are treated as black-box instructions, i.e., provided the parameters are valid and the timing satisfies prerequisites, the HIL bench is expected to complete the corresponding instructions.

## 2.2 ECUs Based on AUTOSAR Classic Platform Architecture

We focus on ECUs with the layered architecture of the AUTOSAR (AUTOMotive Open Software Architecture) Classic Platform [8], as shown in Fig. 2. At the highest level, ECU software is typically divided into the *Application Layer*, the *Run-Time Environment* (RTE), and the *Basic Software* (BSW).

**Application Layer.** It consists of multiple application software components (SWCs) that implement specific vehicle functions. Each SWC contains several *runnables*, which are the atomic units of behavioral logic that can be scheduled. Runnables are responsible for executing specific algorithmic logic (e.g., periodic filtering, state machine stepping, or diagnostic request processing). They are triggered by timing events or data-received events, and are dispatched by the RTE into a specific OS task, along with processing functions from the BSW which have the same periodicity. An OS task constitutes the fundamental unit of execution scheduled by the operating system. It corresponds to a task entry function, commonly seen as TASK(TaskName) in real-world projects. The task-runnable mapping determines the actual run-time execution timeline and preemption relationships.

**Runtime Environment (RTE).** It serves as the middleware layer that facilitates communication between the application layer and the BSW. It abstracts the underlying hardware and BSW services, offering standardized interfaces for SWCs to interact with each other and with lower-level services. Moreover, it implements the mapping between runnables and OS tasks.

**Basic Software (BSW).** It is further divided into three main parts, i.e., the service layer, the ECU abstraction layer, and the microcontroller abstraction layer (MCAL). The service layer encompasses a suite of standardized services that provide essential functionalities to the application layer, including system services, communication stacks, diagnostic services and MCAL drivers. The ECU abstraction layer offers a

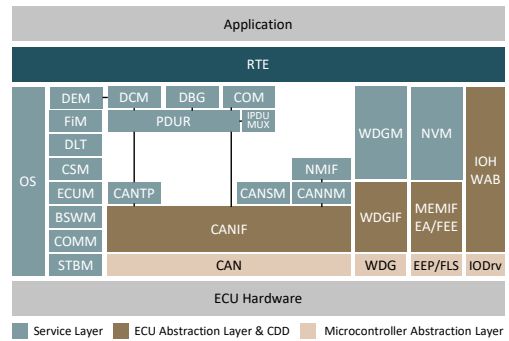


Fig. 2. The AUTOSAR Classic Platform Architecture



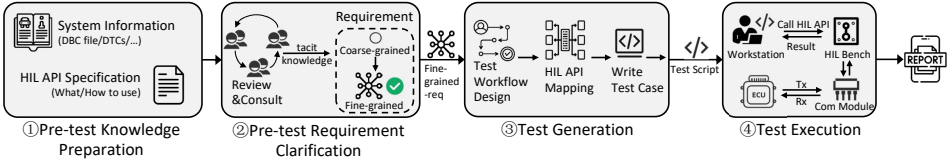


Fig. 3. Typical Workflow for ECU Functional Testing on HIL

hardware-independent interface to the upper layers. The MCAL directly interfaces with the microcontroller hardware, providing low-level drivers for peripherals. Additionally, complex device drivers (CDD) support specialized hardware with strict timing constraints.

In the context of ECUs based on the AUTOSAR classic platform architecture, we define **module** as a cohesive, end-to-end functional chain on an ECU that spans multiple AUTOSAR layers and is identifiable in both configuration and runtime execution. It represents the system-level behavior realized by SWCs orchestrating BSW services via RTE. For example, a CAN communication module involves CAN, CAN Interface (CANIF), PDU Router (PDUR), and COM services in the BSW layer, RTE for message routing, SWCs like CAN\_Management in the application layer for processing the transmitted/received data, and other dependent services.

### 2.3 Current Workflow for ECU Functional Testing

As illustrated in Fig. 3, the current workflow for functional testing of automotive ECUs on HIL benches generally involves the following four main phases.

**Pre-test Knowledge Preparation.** Test engineers must comprehend both the ECU under test and the HIL environment. They not only consult various documents, e.g., vehicle configurations (Diagnostic Trouble Codes, i.e., DTCs) and communication matrices (Database for CAN, i.e., DBC file) to build a holistic understanding of the ECU, but also refer to HIL API specifications to understand functional boundaries, constraints, parameter semantics, and timing/prerequisite conditions of THPs.

**Pre-test Requirement Clarification.** Test engineers need to liaise with system architects and developers to resolve ambiguities in the natural language requirements, turning a coarse-grained requirement into a fine-grained one. Through testability reviews, they clarify abstract functional descriptions into specific behavioral scenarios, explicitly defining preconditions, trigger events, expected behaviors and the corresponding tolerance/threshold, configuration values, and test oracles.

**Test Generation.** Test engineers typically organize test activities into test steps following the sequence of “Initialization → Stimulus → Observation → Verification → Cleanup”, map these steps to specific THP invocation sequences, and implement the test logic in script languages like Python.

**Test Execution.** After script implementation, test engineers deploy the scripts onto the HIL bench for execution. They monitor the execution and analyze the results. If a failure arises, they debug the scripts, adjust test parameters, or even revisit earlier phases to clarify requirements or modify test scripts; or they deep into the source code of ECU to analyze the root cause of the functional failure.

### 2.4 Motivation

Pre-test knowledge preparation faces two critical hurdles. First, test scripts often require adaptation due to HIL bench heterogeneity [24], which imposes high learning and migration costs. While some commercial benches (e.g., dSPACE) have adopted the ASAM XIL standard [15] (i.e., an interface standard defined to decouple test automation frameworks from test execution systems), proprietary in-house HIL benches contend with incompatibilities regarding THP naming conventions, parameter contracts, and usage patterns, forcing testers to delve into bench source code to master THPs. Second, for LLMs, the lack of the knowledge for all valid THPs and their semantic constraints often leads to hallucinations during test generation. For example, models may fabricate non-existent

interface names, misuse parameters, or omit necessary initialization steps, resulting in syntactically or semantically non-executable scripts [4, 40]. Therefore, establishing a THP knowledge base via a generalizable construction method serves as the first cornerstone for AUTOHIL, ensuring test script executability across heterogeneous HIL benches.

For requirement clarification, early-stage requirements often remain implementation-agnostic. Details such as internal state machines, filtering strategies, error recovery paths, and parameter thresholds are refined progressively by testers based on individual experience, or consulting with developers and inspecting the ECU source code. This introduces tacit domain knowledge [6], which is often attached to the accumulated experience of domain experts. Leaving these ambiguities unresolved exacerbates the impact of requirement volatility, which continuously affects the stability of test scripts, adding overhead to test design and maintenance [31]. For LLMs, the absence of such tacit domain knowledge in the requirements can lead to incomplete or incorrect test logic generation. Therefore, augmenting requirements with tacit domain knowledge extracted from ECU source code becomes the second cornerstone for AUTOHIL to ensure functional correctness of test scripts.

For test generation, there exist two challenges. Regarding automation capability, current industrial practice lacks frameworks for automated test generation. Regarding granularity alignment, the logical steps decomposed by testers often do not map directly to THPs, making it difficult to translate high-level intent into executable primitives. Therefore, an end-to-end automation framework that systematically aligns test steps with THPs is the overarching objective of AUTOHIL.

### 3 Methodology

#### 3.1 Overview

**3.1.1 Assumptions.** Our design is independent of any specific OEM or HIL platform. To delineate the scope of applicability and ensure the feasibility of the automation workflow, we premise our design on two critical assumptions. First, we assume the testers have full access to both the THP information and the source code of ECU under test. We target in-house testing during the development phase, rather than black-box validation. For commercial HIL benches, this means the official THP documentation is available. For proprietary in-house benches, it implies direct access to the underlying source code. Regarding the ECU, we assume access to the C source code, or at least the code segments related to the module under test. This assumption is consistent with the industrial practice, where development and testing responsibilities increasingly converge [27], making it common for testers to be familiar with both THPs and the source code [46]. Second, we assume the ECU under test adheres to the AUTOSAR classic platform architecture, which is widely adopted in the automotive industry. This assumption allows us to leverage the clear functional boundaries and execution models defined by the AUTOSAR standard, facilitating the identification of modules.

**3.1.2 Framework Pipeline.** Holistically, AUTOHIL takes the following artifacts as inputs.

- A set of system functional requirements  $R = \{r_1, r_2, \dots, r_n\}$  to be tested, where each  $r_i$  is a natural language description of a specific functional requirement.
- A set of THPs  $T = \{t_1, t_2, \dots, t_m\}$ , where each  $t_j$  is a THP function which can be invoked directly in test scripts to simulate specific behaviors.
- The source code of the ECU under test  $S$ , written in AUTOSAR-compliant C language.
- Configurations  $C$ , including not only the ECU configurations (e.g., DTC definitions, timing/delay constraints, and parameter ranges), but also the HIL configurations (e.g., hardware I/O mappings, plant model parameters, and fault injection assignments).

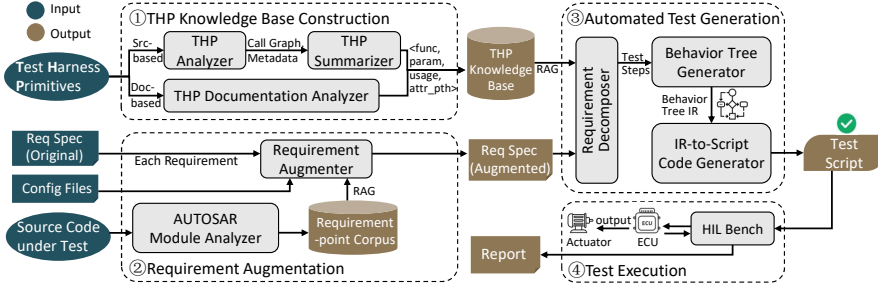


Fig. 4. Framework Overview of AutoHIL

The goal of AUTOHIL is to automatically generate a set of test scripts  $P = \{p_1, p_2, \dots, p_n\}$ , where each  $p_i$  is a test script that verifies the requirement  $r_i$  in  $R$ , establishing an explicit one-to-one mapping  $r_i \xrightarrow{\text{AutoHIL}} p_i$ . To achieve this, AUTOHIL employs a four-stage pipeline, as shown in Fig. 4.

To comprehend THPs and prevent hallucinations, the first stage is *THP Knowledge Base Construction*. It either structures the THP documentation (*THP Documentation Analyzer*), or performs static analysis (*THP Analyzer*) and summarization (*THP Summarizer*) on the THP source code, to build a THP knowledge base  $K_{\text{THP}}$ , where  $K_{\text{THP}}(t_j)$  stores the functionality, parameters, usage patterns, and attribute access path of the THP function  $t_j$ , enabling effective retrieval during later stages.

To bridge the tacit domain knowledge gap in functional requirements, the second stage is *Requirement Augmentation*. It conducts AUTOSAR-aware static analysis on the ECU source code  $S$ , and produces semantic summaries via LLM for each function in each module (defined in Sec. 2.2) to understand the implementation details. Then, it utilizes LLM to extract logical behaviors into requirement points, which carry the tacit domain knowledge (*AUTOSAR Module Analyzer*). AUTOHIL organizes them into a requirement point corpus  $Q$ . Using retrieval-augmented generation (RAG) techniques, AUTOHIL then combines  $Q$  with configurations  $C$  to enrich the original requirements  $R$  into augmented requirements  $R' = \{r'_1, r'_2, \dots, r'_n\}$  (*Requirement Augmenter*).

To achieve end-to-end automation, the third stage is *Automated Test Generation*, where AUTOHIL leverages the THP knowledge base  $K_{\text{THP}}$  and the augmented requirements  $R'$  to automatically generate test scripts  $P$ . For each augmented requirement  $r'_i$ , AUTOHIL leverages LLM together with  $K_{\text{THP}}$  to decompose  $r'_i$  into a sequence of test steps  $\text{Seq}_i$  (where each step is instantiated as a specific THP) along with the control structure and variable definition (*Requirement Decomposer*). Subsequently, AUTOHIL generates a behavior tree  $B_i$ , which captures the hierarchical structure of test steps and control structure (*Behavior Tree Generator*), and then decodes it into a HIL test script  $p_i$  (*IR-to-Script Code Generator*). It can be formally represented as:  $r'_i, K_{\text{THP}} \rightarrow \text{Seq}_i \rightarrow B_i \rightarrow p_i$ .

Finally, AUTOHIL executes  $P$  on the HIL bench. Each test script contains assertions as an oracle to produce a verdict  $v_i$  (pass/fail), which serves as the primary output of the testing report.

### 3.2 THP Knowledge Base Construction

The goal of this stage is to transform heterogeneous and largely unstructured THP artifacts (source code or documentation) into a machine-readable and semantically explicit knowledge base  $K_{\text{THP}}$ . For each THP  $t_j \in T$ , we define its knowledge  $K_{\text{THP}}(t_j)$  as a 4-tuple  $\langle \text{func}_j, \text{param}_j, \text{usage}_j, \text{attr\_pth}_j \rangle$ , where  $\text{func}_j$  is functionality description,  $\text{param}_j$  denotes the input/output parameter syntax and semantics,  $\text{usage}_j$  is the usage pattern, and  $\text{attr\_pth}_j$  is the attribute access path (e.g., in the expression  $A.b.c.\text{func}()$ , the sequence  $A.b.c$  constitutes the attribute access path of  $\text{func}$ ). To accommodate the diversity of HIL ecosystems, we consider two representative settings, in-house HIL benches,

where THP semantics are primarily encapsulated with detailed implementations (Sec. 3.2.1 and 3.2.2), and commercial HIL benches, which are typically delivered without source code (Sec. 3.2.3).

**3.2.1 Source-Based Construction: THP Analyzer.** On in-house HIL benches, the THP set  $T = \{t_1, t_2, \dots, t_m\}$  is frequently heavily wrapped, involves implicit state machines, and lacks complete documentation. Thus, relying solely on function names or sparse comments is insufficient for reliable automated test generation. The *THP Analyzer* is responsible for recovering the reachable call structure *CallGraph* and key static metadata *Meta* of each THP from the implementation code, providing a structural backbone and grounded evidence for subsequent semantic summarization.

**Self-Contained Sub-Package Extraction.** We begin from the project interface layer, i.e., the small set of wrapper modules that test scripts import directly to access bench operations. All functions exported by this layer are considered as script-callable THPs, forming the THP set  $T$ . Using each  $t_j \in T$  as a starting point, we recursively traverse module import dependencies by analyzing import statements. This process yields a HIL project self-contained sub-package  $pkg_{THP}$ , which can be viewed as a code closure reachable from the test script perspective. Confining analysis to  $pkg_{THP}$  reduces computational overhead and minimizes interference from irrelevant library code.

**Call Graph Construction.** On  $pkg_{THP}$ , we utilize Joern [45] to generate a code property graph (CPG) that combines information from the abstract syntax tree (AST), control-flow graph (CFG), and program dependence graph (PDG). Joern explicitly maintains caller and callee for each function node, thus we derive the call graph  $G = (V, E)$  directly from the CPG, where each vertex  $v \in V$  represents a function, and a directed edge  $(u, v) \in E$  exists if function  $u$  calls function  $v$ . For each THP function  $t_j \in T$ , we perform a depth-first search (DFS) on  $G$  rooted at  $t_j$  to obtain its specific call sub-graph. We then construct a unified *CallGraph*  $= (V', E')$  by merging these individual sub-graphs, specifically by consolidating identical function nodes and edges shared across different sub-graphs. By doing so, we enable a global analysis of  $pkg_{THP}$ 's call structure through a single graph traversal, rather than redundantly processing fragmented sub-graphs.

**Static Metadata Acquisition.** We also extract essential static metadata *Meta* from CPG. For each function node  $v \in V'$ ,  $Meta(v) = \langle sign, file\_pth, line\_num, macro \rangle$ ; if a function node corresponds to a THP  $t \in T$ , its metadata is  $Meta(t) = \langle sign, file\_pth, line\_num, macro, attr\_pth \rangle$ . Attributes like function signature (*sign*), source file path (*file\_pth*), and line number (*line\_num*) are directly queried from CPG nodes, in order to locate and retrieve the source code snippets for the *THP Summarizer* (Sec. 3.2.2). Used macros (*macro*), represented as calls in Joern, are recursively resolved to capture both their semantic names and logic code, ensuring that *THP Summarizer* comprehends the functional logic. Attribute access paths (*attr\_pth*) are resolved by mapping each THP to its subclass, handling inheritance and reconstructing the full instantiation chain.

**3.2.2 Source-Based Construction: THP Summarizer.** We next compress the reachable code closure into a semantic representation  $K_{THP}$  that can be consumed efficiently and reliably by the LLM. Feeding the entire call graph directly to an LLM causes context overflow, introduces redundancy from intermediate wrappers, and distracts the LLM with excessive low-level details. Therefore, the *THP Summarizer* employs a bottom-up hierarchical summarization strategy; for each THP  $t_j \in T$ , it summarizes *func<sub>j</sub>*, *param<sub>j</sub>*, and *usage<sub>j</sub>*, stores them together with *attr\_pth* in *Meta*, forming  $K_{THP}(t_j) = \langle func_j, param_j, usage_j, attr\_pth_j \rangle$ . Finally,  $K_{THP}$  is indexed for semantic retrieval, enabling RAG-style access and constraint-aware validation in later test generation (Section 3.4).

**Out-Degree-Based Reverse Topological Layering.** We apply reverse topological sorting to *CallGraph* to organize functions into layers and establish a strictly bottom-up processing order, ensuring that for any acyclic edge  $(u, v) \in E'$ , the summary of callee  $v$  is available when generating caller  $u$ 's summary. To accommodate cyclic dependencies common in embedded software, we detect strongly connected components (SCCs) via DFS, and summarize each SCC as a whole.

### LLM-Based Function-Level Summary Generation.

We traverse the topological layers from bottom to top. For an acyclic function  $v$ , the *THP Summarizer* provides the LLM with four types of context: the function signature (*sign*), source code snippet derived from *file\_pth* and *line\_num*, macro definitions (*macro*), and summaries of its direct callees. Then, the LLM is instructed by Fig. 5 to generate the summary. For cycles, we provide the implementation of all functions in the SCC along with their mutual call relationships, generate a coherent SCC-level description first, and then generate the summary for individual functions. After obtaining summaries for all functions in *CallGraph*, the knowledge base exposes only entries in the THP set  $T$ . Summaries of non-THP nodes serve as intermediation to stabilize hierarchical summarization, but are not indexed as retrieval targets.

Finally, we serialize each  $K_{THP}(t_j)$  into a structured text document. These documents are processed by a text embedding model (e.g., Google GenAI embeddings) to generate high-dimensional vector representations, which are then stored in a vector database (e.g., ChromaDB). This vector index enables semantic retrieval, allowing the system to match natural language test requirements with the most semantically relevant THP functions based on vector similarity.

**3.2.3 Documentation-Based Construction.** For commercial HIL benches (e.g., dSPACE), we directly extract the  $K_{THP}$  tuples from their systematic documentation, ensuring consistency with the source-based method. Vendor APIs typically provide structured references where functionality descriptions (*func*) and input/output parameter syntax and semantics (*param*) are explicitly stated. Crucially, the documentation organizes APIs by class or module hierarchy, allowing us to directly resolve the attribute access path (*attr\_pth*) without complex dependency tracing, and often includes standard code snippets for common tasks, which we parse to extract valid usage patterns (*usage*).

### 3.3 Requirement Augmentation

We first extract source-grounded evidence from the AUTOSAR ECU source code  $S$  to construct a requirement point corpus  $Q$ , and then we augment each raw requirement with source-grounded evidence from  $Q$  and configuration facts from  $C$ . The output is an augmented set  $R' = \{r'_1, \dots, r'_n\}$ . Each  $r'_i$  should be test-ready, which includes preconditions (GIVEN), a trigger event (WHEN), an expected behavior (THEN) and its corresponding tolerance/threshold (WITHIN), as well as concrete configuration values (CONFIG) and a test oracle (CRITERIA).

**3.3.1 AUTOSAR Module Analyzer.** The *AUTOSAR Module Analyzer* follows the same general paradigm as *THP Analyzer* and *THP Summarizer* (i.e., static analysis plus hierarchical summarization), but the cognitive objectives differ fundamentally. While the latter focuses on the interface specifications of the bench, producing  $K_{THP}$  to guide “how to drive the HIL bench”, the former targets the behavioral logic of the ECU, and acts as a knowledge distillation pipeline that transforms raw, unstructured ECU source code  $S$  into a structured requirement-point corpus  $Q$  to reveal “what the ECU should do under specific conditions”. The resulting corpus  $Q$  serves as a source-grounded evidence pool that bridges the gap between natural language requirements and code implementation.

**ECU Source Code Analysis.** Following the definition of an ECU module as a functional chain in Sec. 2.2, we identify specific ECU modules explicitly listed in the system configuration  $C$ . For each ECU module  $M$ , we first identify its entry point *Entry\_M*. Specifically, *Entry\_M* is typically the entry

**[System]** You are an expert in code analysis, specializing in generating function summaries. Please generate summary for each function.

**[User] Input:**

- Function Signature: {sign}
- Source Code: {code snippet (derived from file\_pth and line\_num)}
- Macro Definitions: {macro}
- Callee Summaries: {callee summaries}

**Output:** Generate with following details:

- a. Functionality description
- b. Input/Output Parameters (name, meaning, default, special constraints like valid ranges)
- c. Usage Pattern (basic usage, complex usage)

Fig. 5. Prompt Template for THP Summarizer



function of the OS task responsible for scheduling the module's behavior (e.g., `TASK(TK_5MSEC)` for the CAN communication module), or the main function of the runnable directly implementing the logic (e.g., `CAN_management.c` for the CAN communication module). Starting from *Entry\_M*, we recursively traverse dependencies by analyzing `#include` statements. This process collects all directly and indirectly referenced source files and headers to construct a self-contained sub-package *ecu\_pkg<sub>M</sub>* for *M*, isolating the code closure of *M* from the massive ECU codebase for focused analysis. Similarly, after call graph construction and metadata acquisition, we obtain *CallGraph<sub>M</sub>* for *M* and metadata *Meta(v)* =  $\langle \text{sign}, \text{file\_pth}, \text{line\_num}, \text{macro} \rangle$  for each function *f* in *CallGraph<sub>M</sub>*, laying a structural foundation for subsequent function summarization and requirement point extraction.

### LLM-Based Hierarchical Summarization for Module Semantics.

Given *CallGraph<sub>M</sub>*, we reuse the “reverse topological layering and bottom-up summarization” process (Sec. 3.2) to generate a semantic summary for each function *f* in *CallGraph<sub>M</sub>* via the prompt in Fig. 6. We focus on internal semantics to uncover testable behaviors hidden in the implementation. Formally, the summary is defined as a tuple *Summary<sub>M</sub>*(*f*) =  $\langle \text{role}, \text{key\_vars}, \text{trigger}, \text{logic\_pts} \rangle$ , where *role* is the high-level responsibility of *f* within *M*; *key\_vars* is key variables read or written by *f* along with their business semantics (e.g., counters, torque commands); *trigger* records the events that activate specific behaviors (e.g., setting/clearing a DTC, switching to a fail-safe mode); and *logic\_pts* highlights the significant logic relevant to testing, such as concrete boundary checks, specific error code assignments, or loop termination conditions.

**Requirement Point Corpus Construction.** A requirement-point *req\_pt* is defined as a coherent logical behavior inferred from the implementation, formulated as *req\_pt* =  $\langle \text{traceability}, \text{desc}, \text{trigger}, \text{expected}, \text{completeness}, \text{confidence} \rangle$ , where *traceability* links to the specific source code function; *desc* is a description following the *GIVEN-WHEN-THEN-WITHIN* paradigm to make logical behavior explicit; *trigger* and *expected* are the trigger condition and expected outcome of the behavior; and *completeness* and *confidence* indicate information completeness and model confidence.

We synthesize requirement points for a module *M* by performing chain-of-thought on condensed representation of source code (structural context *CallGraph<sub>M</sub>* and semantic context *Summary<sub>M</sub>*). The LLM is prompted to first analyze *CallGraph<sub>M</sub>* to identify the module skeleton (function nodes and call edges), then fuse function summaries to understand source code functionalities, and finally extract independent requirement points and convert them into natural language specifications.

The requirement point corpus *Q* lays the foundation for requirement augmentation in two ways. First, a single requirement may correspond to multiple fine-grained *req\_pts*, thus we assemble these *req\_pts* to fill in the broad description with concrete details, making the requirement test-ready. Second, a *req\_pt* may reveal missed logic, which can supplement the original requirements.

**3.3.2 RAG-Based Requirement Augmenter.** *Requirement Augmenter* enriches each original requirement  $r_i \in R$  into an augmented requirement  $r'_i$ . We use RAG to inject implementation evidence and configuration facts into the requirement description. We define the augmented requirement  $r'_i$  as:

$$r'_i = \text{Augment}(r_i, \mathcal{X}_i) = (\text{GIVEN}, \text{WHEN}, \text{THEN}, \text{WITHIN}, \text{CONFIG}, \text{CRITERIA}),$$

**[System]** You are an expert in code analysis, specializing in generating function summaries. Please generate summary for each function.

**[User] Input:**

- Function Signature: {sign}
- Source Code: {code snippet (derived from file\_pth and line\_num)}
- Macro Definitions: {macro}
- Callee Summaries: {callee summaries}

**Output:** Generate with following details:

- a. Role (functionality, responsibility)
- b. Key Variables (name, semantics)
- c. Trigger Conditions (specific behaviors)
- d. Logical Points (significant logic in the func)

Fig. 6. Prompt Template for LLM-Based Hierarchical Summarization of Module Semantics

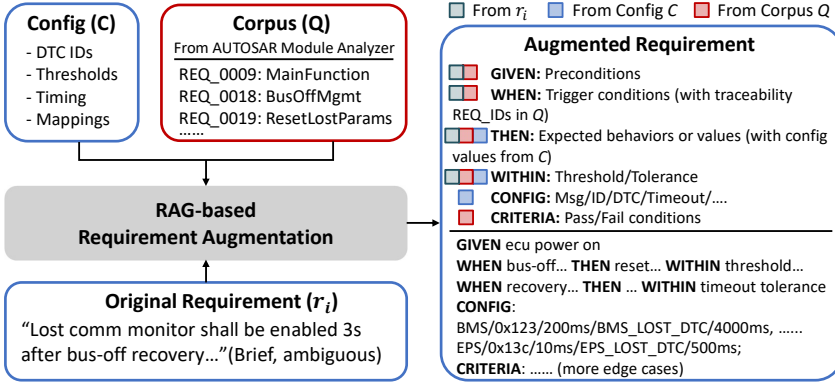


Fig. 7. Illustrative Example of Requirement Augmentation

where  $r'_i$  contains fields used in test specifications: **GIVEN** describes preconditions from  $r_i$ ; **WHEN** specifies trigger conditions; **THEN** outlines expected behaviors; **WITHIN** defines tolerances or thresholds for expected behaviors; **CONFIG** lists concrete configuration parameters and values from  $C$ ; and **CRITERIA** establishes the test oracle with pass/fail assertion.  $\mathcal{X}_i$  is a context constructed around  $r_i$  that explicitly represents the extra information used for augmentation:

$$\mathcal{X}_i = \langle Evi_i, Fact_i \rangle, \quad Evi_i = \text{Evidence}(r_i, Q), \quad Fact_i = \text{Facts}(Evi_i, C).$$

Here,  $\text{Evidence}(r_i, Q)$  selects an evidence set  $Evi_i$ , consisting of multiple *req\_pts* from the corpus  $Q$ , which identifies the implementation logic corresponding to  $r_i$ , through a two-stage retrieval process. First, we use vector similarity to fetch candidate requirement points from  $Q$ . Second, we re-rank these candidates based on their *completeness* and *confidence* attributes. The resulting  $Evi_i$  serves as a semantic bridge that maps the abstract intent of  $r_i$  to concrete **GIVEN-WHEN-THEN-WITHIN** behaviors derived from implementation.  $\text{Facts}(Evi_i, C)$  represents configuration data (e.g., DTC definitions, thresholds, or enums) retrieved from  $C$  that relates to  $Evi_i$ . This helps ground the LLM's result in executable engineering details (**CONFIG**). Given  $r_i$  and  $\mathcal{X}_i$ , *Requirement Augmenter* employs LLM to generate **GIVEN-WHEN-THEN-WITHIN** and **CONFIG** by combining  $Evi_i$  and  $Fact_i$ , and to derive pass/fail **CRITERIA** based on expected behaviors and tolerances in  $Evi_i$ . Through augmentation, each  $r'_i \in R'$  becomes test-ready.

**3.3.3 Illustrative Example.** To illustrate our source-grounded requirement augmentation, we use the CAN communication module as an example. We first parse the ECU source code and identify the main execution entry of this module, which is a 5-ms periodic OS task function TASK (TK\_5MSEC). Static analysis then recovers a self-contained sub-package *ecu\_pkg\_CAN* that spans *CANMGR-RTE-COM-PDUR-CANIF-CAN* services, along with necessary supporting services such as OS and NVM. Then, we obtain the call graph of *ecu\_pkg\_CAN*, and maintain a set of function metadata. We then traverse the call graph bottom-up to generate function summaries. By feeding both the topological structure and semantic summaries into the synthesis engine empowered by LLM, we construct a corpus of 102 extracted requirement points. Finally, as shown in Fig. 7, utilizing the corpus and the module's configurations as a unified retrieval base, we execute RAG-based augmentation on the original requirement, and the augmented requirement adds concrete triggers and constraint details to the original one. For example, the augmenter adds logic "WHEN bus-off THEN reset counters WITHIN time threshold" from  $Q$ , while the original requirement only focuses on the recovery logic; adds concrete DTCs, cycle time and timeout for CAN messages from  $C$ ; and also encompasses more

edge cases in pass/fail criteria. We annotate each added element with its traceable source in the configurations  $C$  or the requirement point corpus  $Q$ .

### 3.4 Automated Test Case Generation

Given the constructed THP knowledge base  $K_{\text{THP}}$  and the augmented requirements  $R'$ , we first use *Requirement Decomposer* to decompose each augmented requirement  $r'_i \in R'$  into a sequence of test steps  $\text{Seq}_i$  (where each step is instantiated as a specific THP) along with the control structure and declared variables. Then, we leverage *Behavior Tree Generator* to transform  $\text{Seq}_i$  into a behavior tree  $B_i$  as an intermediate representation (IR), which is finally compiled into an executable test script  $p_i$  that can be directly run on the HIL bench through *IR-to-Script Code Generator*.

**3.4.1 Requirement Decomposer.** Taking  $r'_i$  and  $K_{\text{THP}}$  as inputs, LLM-empowered *Requirement Decomposer* generates a sequence of abstract test steps  $\text{Stp}$  along with the control structure (that organizes these steps) and declared variables (that are used in conditional checks in the control structure). Steps in  $\text{Stp}$  include: (i) initialization, which establishes the preconditions defined in GIVEN (power and environment setup); (ii) stimulus, which produces the trigger events defined in WHEN (signal injection); (iii) observation, which captures the actual behavior or state values of the ECU (periodic monitoring); (iv) verification, which compares the observation against expected outcomes in THEN and tolerance in WITHIN, asserting compliance with CRITERIA (checking states or DTCs); and (v) cleanup, which restores safety states to prevent side effects on next tests.

Subsequently, *Requirement Decomposer* aligns each abstract test step  $s \in \text{Stp}$  with a concrete THP. We map the natural language description of  $s$  into a query vector, and then calculate the cosine similarity between the query and the functionality descriptions of all available THPs in  $K_{\text{THP}}$ , and retrieve the most similar THP. The arguments passed to the THP are instantiated by extracting logical values from the step description. Finally, it outputs  $\text{Seq}_i$  as a sequence of instantiated THPs with arguments, declared variables (derived from CONFIG) for global reuse or loop iteration, and the control structure of  $\text{Seq}_i$  to explicitly organize the logical flow of the THP sequence.

**3.4.2 Behavior Tree Generation.** Behavior trees can represent complex behavior as a hierarchy, making test logic easier to express. Thus, they have been widely adopted for test case specification, scenario-driven testing, and test generation [32, 39], with capabilities to extend node types to fit specific needs [19]. In AUTOHIL, we employ behavior tree as an IR between requirements and test scripts, and customize standard behavior tree definitions for HIL testing.

**Behavior Tree Definition Customized for HIL Testing.** The customized node set is categorized into *composite*, *decorator*, and *leaf* nodes. Composite nodes dictate the control flow: a *Sequence* executes child nodes in strict order, failing immediately if any child fails, whereas a *Selector* executes its child nodes sequentially and succeeds immediately upon the first success. Decorator nodes modify the behavior of their single child: a *Condition* acts as a logical guard, executing the child only if a specified predicate is true; a *Loop* repeats execution either for a fixed count or until a termination condition is met; and a *ForEach* iterates over a dataset, executing its child node for each element in the dataset. The execution terminates at leaf nodes, represented solely by *Action* nodes, which encapsulate concrete THP invocations. Each node contains generic identifiers (type, name) alongside type-specific attributes. Specifically, *Action* nodes store the THP name (function) and a dictionary of arguments (args). *Condition/Loop* nodes encapsulate boolean logic, using params and expected fields to store expressions evaluable in Python, or referencing signal-reading functions. *ForEach* nodes define the iteration variable var and the target iterable collection iterable.

**THP-Guided Behavior Tree Instantiation.** To standardize the test script topology, we mandate that the root of the behavior tree  $B_i$  is a *Sequence* node, with its child nodes naturally mapping to the five standard phases of  $\text{Seq}_i$ , i.e., precondition, stimulus, observation, verification, and cleanup. Given

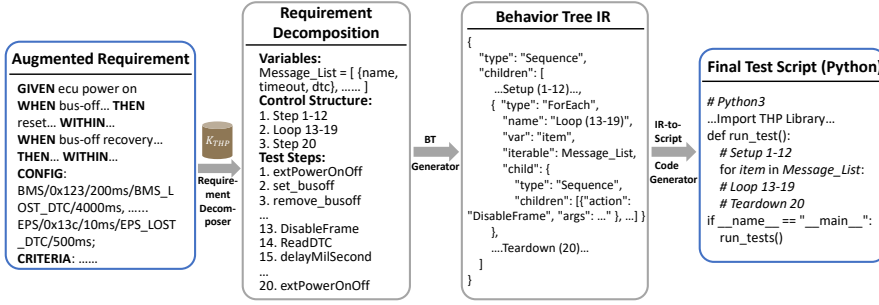


Fig. 8. Illustrative Example of Test Script Generation

$Seq_i$ , LLM-empowered *Behavior Tree Generator* maps the phases to specific nodes; i.e., initialization, stimulus, and cleanup phases are transformed into *Action* nodes with their THPs; observation phase is encapsulated in *Loop* nodes to enforce monitoring durations, which typically nest *ForEach* nodes for iterations over key variables; and verification is translated into *Condition* nodes for assertions.

During behavior tree generation, *Behavior Tree Generator* is prompted to adhere to strict constraints so that  $B_i$  is statically checkable and safe to compile. At the THP level, every *Action*.function must resolve to a key in  $K_{THP}$ . In addition, *Action*.args must follow the parameter definitions in  $K_{THP}$ , including type, unit, and range. At the expression level, we restrict the logic in *Condition* and *Loop* to a safe subset, namely comparisons, Boolean operations, arithmetic, and parentheses. AutoHIL performs immediate structural and type checks on the generated  $B_i$ . Any errors trigger a feedback loop, prompting the *Behavior Tree Generator* to correct  $B_i$  until it passes checks.

**3.4.3 IR-to-Script Code Generation.** Upon checking  $B_i$ , the *IR-to-Script Code Generator* translates it into a HIL test script  $p_i$  with a rule-based transformation and an LLM-based refinement. The transformation rules enforce an isomorphic mapping from behavior tree syntax to Python syntax. The root *Sequence* becomes the top-level test function  $test\_req\_id()$ , with child nodes expanded into sequential code blocks. *Selector* nodes are translated into branches (e.g., if/elif or try/except). *Condition* nodes are converted into if statements or assert checks with Boolean expressions, invoking data-fetching THPs where necessary. *Loop* and *ForEach* nodes map to iteration structures (i.e., for/while), and *Action* nodes are expanded into direct THP function calls. The test function returns True only if all verifications pass without exceptions; otherwise, it logs the context and returns false. After rule-based transformation, we also embed a lightweight LLM-based refinement engine to perform static correction, which is safer and cheaper than on-bench debugging due to the high cost of HIL execution and hardware safety risks. Concretely, the generated test script, along with relevant THP summaries, is provided for the LLM for syntax and semantic check and refinement. The LLM is strictly restricted to performing minimal fixes, such as correcting formatting and argument literals to satisfy declared types and ranges, without altering the control flow or introducing new THPs. This yields a reliable executable test script  $p_i$  for the requirement  $r_i$ .

**3.4.4 Illustrative Example.** We continue with the CAN communication module example to illustrate the test generation process. As illustrated in Fig. 8, given the augmented requirement in Sec 3.3.3, *Requirement Decomposer* generates a sequence of instantiated THPs ( $Seq$ ) and defines  $Seq$ 's key variables and control structure. Specifically, this control structure means that steps 1-12 are sequentially executed, then steps 13-19 are iteratively executed in a loop, and finally step 20 is executed. Then, *Behavior Tree Generator* structures  $Seq$  into a formal behavior tree. For example, the iteration logic covering steps 13-19 is mapped to a *ForEach* node and configured with an iterable attribute pointing to the key variable Message\_List. Its child is a nested *Sequence* node containing

specific *Action* nodes, such as `DisableFrame`, which utilize the current item to execute the steps dynamically for each message configuration in `Message_List`. Finally, *IR-to-Script Code Generator* translates the behavior tree into an executable test script in Python.

## 4 Evaluation

### 4.1 Experimental Setup

Our evaluation is designed to answer the following three research questions.

- **RQ1: Effectiveness Evaluation:** Can AUTOHIL effectively generate ECU test scripts that ensure functional equivalence to test scripts manually written by experts?
- **RQ2: Efficiency Evaluation:** How much time and human effort does AUTOHIL save in generating ECU test scripts compared to manual methods?
- **RQ3: Ablation Study:** What is the impact of each component in AUTOHIL on its performance?

To the best of our knowledge, AUTOHIL is the first automated test generation framework designed for ECU functional testing in HIL environments. Existing automated functional testing for ECU seldom addresses HIL constraints (e.g., bench state, timing, and hardware interaction) and always relies on specific HIL bench. Therefore, we use manually written test scripts by our industrial collaborator as the primary reference for comparison. These manual scripts are historical test artifacts accumulated from the collaborator's industrial projects. Fig. 9 illustrates the hardware configuration in our evaluation. It includes a PCAN interface for CAN communication and a Lauterbach debugger for monitoring states of the ECU.

To answer **RQ1** and **RQ2**, we use two representative ECUs, i.e., an Airbag Control Unit (ACU), which is a classic chassis ECU representing the highest safety integrity level (ASIL-D), and an Active Safety Domain Master (ASDM), which is the most logically complex ECU, responsible for multi-sensor fusion and Advanced Driver Assistance Systems (ADAS) logic. We perform automated test generation based on their functional requirement documents. To answer **RQ3**, we utilize the ACU to evaluate the effectiveness of AUTOHIL with and without THP knowledge base, requirement augmentation, and requirement decomposition. Also, we select four LLMs to assess model sensitivity, including two closed-source models (gemini-2.5-pro, gpt-5) and two open-source models (deepseek-r1, qwen3-32B), comparing the effectiveness of AUTOHIL. The closed-source models are accessed through APIs, while the open-source models support local deployment within the enterprise to avoid data leakage. For the localRAG, we use the Chroma database and gemini text-embedding-004 model.

In our experiments, ACU uses an in-house HIL bench developed by our industrial collaborator. The HIL APIs feature project-level encapsulation, totaling 511 THPs. ASDM is evaluated on a dSPACE SCALEXIO LabBox platform. Its AutomationDesk exposes the test platform as an accessible object and provides built-in Python THP documentation, from which we curate a total of 1,548 THPs. The tested ACU includes 18 modules and 683 requirements, with 567 engineer-verified manual test scripts provided by the collaborator. The tested ASDM includes 19 modules and 1,252 requirements, with 894 engineer-verified manual test scripts. In practice, one manual script often corresponds to multiple requirements. To compare more directly with AUTOHIL (which generates tests on a one-to-one basis), we split the human scripts into requirement-level fragments and align them with specific requirements, ensuring granular consistency with AUTOHIL's output.

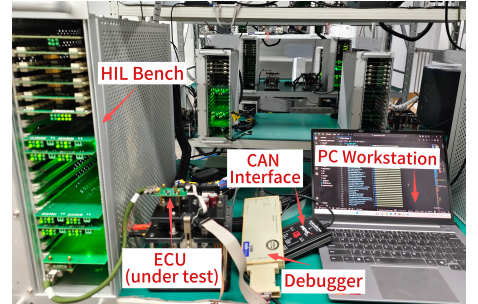


Fig. 9. Hardware Configuration for Evaluation



Table 1. Executability of Generated Test Scripts

| Total                  | ACU (683) |            |        | ASDM (1,252) |            |        |
|------------------------|-----------|------------|--------|--------------|------------|--------|
|                        | #Excluded | #Remaining | Ratio  | #Excluded    | #Remaining | Ratio  |
| <b>Syntactic-Level</b> | 19        | 664        | 97.21% | 21           | 1,231      | 98.32% |
| <b>Semantic-Level</b>  | 26        | 638        | 93.41% | 122          | 1,109      | 88.58% |
| <b>Oracle-Level</b>    | 21        | 617        | 90.33% | 91           | 1,018      | 81.31% |

## 4.2 Effectiveness Evaluation (RQ1)

To evaluate the effectiveness of our framework, we first evaluate whether scripts are executable, then assess functional equivalence between the generated scripts and expert-written manual scripts, and finally compare their defect detection capability.

**4.2.1 Executability Evaluation.** We evaluate the executability of a test script by three levels.

- **Syntactic-Level:** The script is free from syntax errors and can be parsed by the interpreter.
- **Semantic-Level:** The script runs without errors, catches no exceptions, and exits normally.
- **Oracle-Level:** The script is runnable and contains a test oracle (assertions or logical comparisons), which determines whether the requirement is met.

As shown in Table 1, the *#Excluded* column reports the number of scripts that fail to meet the criterion of each level. Ultimately, 90.33% and 81.31% of the generated scripts for ACU and ASDM achieve all executability criteria, respectively. After excluding sporadic issues caused by bench instability or real-time jitter, we categorized failed cases into four primary causes. (1) **Type Mismatches Induced by Hallucination.** When natural language is translated into the behavior tree without strict formalization, the rule-based conversion from behavior tree to script code encounters exceptions due to incorrect data types. This typically occurs during the compilation phase, failing the syntactic-level check. (2) **Incorrect THP Selection.** The script invokes a similar THP that shares a functional purpose but utilizes a different data path. This results in the bench failing to establish the expected interaction with the ECU or reading real-time data from an incorrect channel/signal. This typically occurs during the semantic-level check, and is more prevalent in in-house HIL scenarios and manifests as read/write failures, timeouts, or abnormal exits during run-time. (3) **Timing and Resource Management Errors.** These errors fundamentally stem from the asynchronous mismatch between the sequential execution of test scripts and the real-time response of the HIL system. Specific examples include failing to establish security access (leading to permission denial) or incorrect release of handles (e.g., communication channels, sessions, access objects), resulting in channel lockups. This usually occurs during the semantic-level check. (4) **Absence of Explicit Oracles.** The script runs and exits successfully but lacks assertions, threshold/tolerance judgments, or traceable pass/fail outputs to compare against expected behaviors. Such scripts cannot provide a conclusion regarding the requirements and are therefore not counted as final passed scripts.

The reason for the discrepancy between the results of ACU and ASDM lies in the differences in interface granularity and the level of encapsulation of the benches. The in-house HIL bench utilized for ACU features a higher degree of encapsulation regarding interaction, I/O, and verdict logic (test oracles) are often wrapped into unified THPs. Consequently, THPs execute actions more directly, which results in reduced script orchestration complexity and fewer omissions in cleanup procedures, leading to superior semantic-level executability. Conversely, while the THPs for ASDM (based on dSPACE AutomationDesk) are more standardized and atomic, it is hard to simulate complex scenarios. LLM is required to autonomously orchestrate multiple THPs to achieve a single functional goal, such as diagnostic sessions, wait conditions, timeout controls, and cleanup procedures. Any incompleteness in the cleanup phase of a step leaves the HIL bench in a residual state, causing cascading

Table 2. Execution Results of ACU Scripts

| Total=617 |                         | Pass                           | Fail |
|-----------|-------------------------|--------------------------------|------|
| Manual    | 585                     | 32                             |      |
| Generated | 529<br>TN: 522<br>FN: 7 | 88<br>TP: 25+39(new)<br>FP: 24 |      |

| Manual    | Pass | Fail |
|-----------|------|------|
| Generated | 522  | 7    |
| Pass      | 522  | 7    |
| Fail      | 63   | 25   |

Table 3. Execution Results of ASDM Scripts

| Total=1,018 |     | Pass              |     | Fail                      |  |
|-------------|-----|-------------------|-----|---------------------------|--|
| Manual      |     | 963               |     | 55                        |  |
| Generated   | 867 | TN: 824<br>FN: 43 | 151 | TP: 12+25(new)<br>FP: 114 |  |

|           |        | Manual | Pass | Fail |
|-----------|--------|--------|------|------|
| Generated | Manual |        |      |      |
| Pass      |        |        | 824  | 43   |
| Fail      |        |        | 139  | 12   |

errors in subsequent steps or even disrupting the following test scripts, thereby degrading semantic-level executability. It is important to emphasize that in the ASDM scenario, errors in THP selection were relatively rare due to the unified object models and interface naming conventions; the performance gap stems primarily from the higher implementation burden associated with orchestrating complex sequences, managing timing and states, and constructing test oracles.

**4.2.2 Functional Correctness Evaluation.** To assess whether the automatically generated test scripts achieve the same testing efficacy as expert-written manual scripts, we incorporate manual sampling by experts to verify the functional correctness of the generated scripts. Specifically, we invite seven experts from our industrial collaborator. They include six senior test engineers and one software engineer. The six test engineers cross-review the test logic of generated scripts against manual scripts, and classify each case into three categories; i.e., (1) Type A: fully equivalent, where key steps, THP choices, and execution order are identical; (2) Type B: functionally equivalent but with different paths, where different THP combinations or step orders are used but the same objective is achieved; and (3) Type C: logically non-equivalent, where there are logic errors, missing key steps, or failure to achieve the intended test objective. The software engineer additionally checks cases in which requirement augmentation introduces implementation knowledge. Because Type B requires a careful logic-level review, this cross-review process is time-consuming. Given the experts' limited availability, for ACU, experts randomly sampled 97 generated scripts. The results showed 18 Type A, 69 Type B, and 10 Type C. For ASDM, experts randomly sampled 100 generated scripts. The results showed 79 Type A, 5 Type B, and 16 Type C. In summary, AutoHIL achieves a high functional correctness of 89.69% and 84.00% on ACU and ASDM.

The causes for failure (Type C) were categorized as parameter errors (wrong threshold parameters or missing parameter settings), wrong THP selection, wrong test timing, missing test steps that prevent the test from forming a closed loop, and wrong test methods (e.g., using transient tests rather than ramp tests for boundary thresholds, which can be filtered as noise by smoothing algorithms, or failure in scenario simulation). The higher proportion of Type B in ACU is because the in-house HIL bench contains many similar THPs that share functional intent but differ in data paths. In contrast, for ASDM using dSPACE, a given test objective often corresponds to a more fixed or unique THP combination. Therefore, manual scripts and generated scripts tend to agree on key steps and THP selection, resulting in a higher proportion of Type A and a lower proportion of Type B.

**4.2.3 Defect Detection Capability.** We executed the manual scripts and the generated scripts, recording the final pass/fail verdict for each requirement. Using the manual script verdicts as a baseline, we categorized the results into four types, i.e., Both Pass, Both Fail, Misses (Manual Fail / Generated Pass), and Divergences (Manual Pass / Generated Fail). Crucially, a divergence is not

equivalent to a false positive. We subjected these samples to manual expert review, subdividing them into false positives and new functional defects (i.e., valid failures where the generated script exposed actual implementation issues that the manual scripts failed to cover or trigger.)

Table 2 and 3 report the statistical results of two ECUs. Among the executable 617 test scripts for ACU, we observe 7 misses (i.e., false negatives) and 24 false positives, while the generated scripts discover 39 new functional failures. The overall consistency rate (i.e.,  $(Both\ Pass + Both\ Fail)/Total$ ) is 88.65%, and the accuracy (i.e.,  $(TN + TP)/Total$ ) of defect detection is 94.98%. Among the executable 1,018 test scripts for ASDM, we observe 43 misses and 114 false positives, while finding 25 new functional defects. The overall consistency rate is 82.12%, and the accuracy of defect detection is 84.58%.

False negatives primarily stem from insufficient test intensity or inaccurate stimulation, e.g., selecting overly loose parameter thresholds, failed simulation of operating conditions, lack of ramp testing (causing transients to be filtered by algorithms), and occasional logic defects. These issues cause generated scripts to yield a pass even when implementation anomalies exist. Additionally, insufficient waiting periods or incomplete teardowns during THP composition can lead to state residue, triggering false negatives. False positives, in contrast, arise more often from overly strict thresholds or deviations in verdict logic. Generated scripts tend to set more conservative tolerances and make stricter assumptions regarding timing windows.

The generated scripts exhibit clear defect detection capability on both ECUs. These newly revealed failures have been explicitly confirmed as valid defects by our collaborator and are currently undergoing remediation. We analyzed these new defects and found that they generally align closely with the mechanism of source-grounded requirement augmentation. We categorized these new defects and provided representative defect examples at our replication site [7].

### 4.3 Efficiency Evaluation (RQ2)

We measure efficiency from two complementary angles. First, we compare the end-to-end time required to deliver HIL test scripts. Second, since the framework does not guarantee 100% executability, we quantify the amount of human intervention needed to repair or align non-executable generated scripts, which reflects the real workload in a human-AI co-development setting.

**4.3.1 Time Cost Analysis.** We record the total time required to construct full test scripts for ACU and ASDM, and compare AutoHIL against a conventional manual workflow. The time cost of AutoHIL is measured end-to-end and includes four stages, i.e., prerequisite preparation, requirement augmentation, test generation, and script repair. The manual time is estimated based on surveys with six senior test engineers from our collaborator. For this manual workflow, the total effort is estimated at approximately 20 person-months; based on this estimation, we express the corresponding parallel execution time in workdays (8 hours per day) in Table 4 for easier comparison. As shown in Table 4, AutoHIL completes the first three stages for a single ECU in less than 4.5 workdays, while the manual workflow typically requires 30–33 workdays. This corresponds to a pre-repair speedup of approximately 7.2x–8.9x. Including script repair, AutoHIL requires 37–63.5 hours (fewer than 8 workdays) and achieves a 3.8x–7.1x speedup. For the manual workflow, we assume the test engineers are already familiar with the HIL bench THPs and have internalized the tacit domain knowledge required. Therefore, we do not assign extra time to learning THPs or domain alignment in the manual baseline. In contrast, AutoHIL’s prerequisite preparation phase is an explicit, project-level investment that can be reused and amortized across requirement updates and iterative test development.

**4.3.2 Human Effort (Lines of Change).** Time reduction alone is insufficient if generated scripts require extensive manual rewriting. We therefore quantify human effort using lines of change (LoC) ratio. We compute the code differences between each generated script and the manual script, and

Table 4. Time Cost of Each Phase

| Phase                    | Automation Time (h)      | Manual Time (h)             | Efficiency Improvement |
|--------------------------|--------------------------|-----------------------------|------------------------|
| Prerequisite Preparation | 8 – 10                   | –                           | –                      |
| Requirement Augmentation | 6 – 6.5                  | 56 – 64                     | 8.6x – 10.7x           |
| Test Generation          | 15 – 17                  | 184 – 200                   | 9.4x – 13.3x           |
| Script Repair            | 8 – 30                   | –                           | –                      |
| <b>Total</b>             | 37 – 63.5<br><8 workdays | 240 – 264<br>30–33 workdays | 3.8x – 7.1x            |

we count inserted, deleted, and modified lines. Specifically, we define scripts that require human intervention as the non-executable scripts identified in **RQ1**, i.e., 66 scripts for ACU and 234 scripts for ASDM, as well as executable scripts that still show functional issues, including expert-sampled non-equivalence and post-run mismatches such as false negatives/false positives, i.e., 38 scripts for ACU and 161 scripts for ASDM. Statistics reveal that 74.21% of the generated scripts require no human intervention. For the remaining scripts to be repaired, the overall LoC rate is only 9.98%. Six test engineers jointly participated in the repair process. Under parallel team collaboration, repairing the generated scripts took 8–30 hours, depending on ECU complexity and defect type. On the in-house bench for ACU, a single test script is typically 50-200 lines. On the dSPACE setting for ASDM, THPs are more atomic, and a single test script is typically 300-500 lines. AUTOHIL successfully offloads the generation of over 90% of the boilerplate code and structural test logic.

#### 4.4 Ablation Study (RQ3)

To evaluate how the three key components, i.e., THP knowledge base, requirement augmentation, and requirement decomposition, contribute to AUTOHIL, we conducted ablation experiments on ACU, as reported in Table 5. Additionally, Table 6 shows the impact of different LLMs on AUTOHIL.

**4.4.1 Impact of THP Knowledge Base.** The THP knowledge base serves as the exclusive bridge connecting the LLM (natural language) to the HIL bench (machine instructions). As shown in Table 5, removing the THP knowledge base results in an executability of 0%, indicating the code generated by the LLM is riddled with hallucinated functions that do not exist in the bench environment. Consequently, these scripts fail at the syntactic level and cannot compile or execute.

**4.4.2 Impact of Requirement Augmentation.** In the absence of requirement augmentation, AUTOHIL degrades to a black-box testing approach. Lacking source code context, the LLM generates tests based solely on superficial requirement documents. These scripts typically cover only the simplest I/O logic, which is easy to execute (thus resulting in a high executability) but lacks depth. The low functional correctness on sampled scripts and the small number of newly detected defects indicate a phenomenon of superficial success. Without cognitive access to internal implementation details (e.g., filtering algorithms, implicit state machine transitions, or configuration conflicts) and tacit domain knowledge, the generated scripts execute without error but fail to trigger complex boundary conditions or expose deep system defects.

**4.4.3 Impact of Requirement Decomposition.** Requirement decomposition is designed to manage the complexity of context during generation. Without decomposition, the LLM is forced to process the massive context from the augmented requirement in a single pass. This cognitive overload leads to skipping logical steps, loss of variable definitions, or timing disorders, directly causing runtime errors and lowering the executability. However, because the framework still retains source code knowledge, the generated logic remains more realistic.

Table 5. Results of Ablation Study

|                               | Executability | Functional Correctness | Defect Detection Capability |             |
|-------------------------------|---------------|------------------------|-----------------------------|-------------|
|                               |               |                        | Accuracy                    | New Defects |
| w/o THP Knowledge Base        | 0%            | –                      | –                           | –           |
| w/o Requirement Augmentation  | 89.15%        | 50.51%                 | 46.10%                      | 3           |
| w/o Requirement Decomposition | 79.64%        | 73.19%                 | 74.67%                      | 17          |
| <b>Full</b>                   | <b>90.33%</b> | <b>89.69%</b>          | <b>94.98%</b>               | <b>39</b>   |

Table 6. Performance Comparison of Different LLMs

|                       | Executability | Functional Correctness | Failure Detection Capability |             |
|-----------------------|---------------|------------------------|------------------------------|-------------|
|                       |               |                        | Accuracy                     | New Defects |
| <b>Gpt-5</b>          | <b>90.33%</b> | <b>89.69%</b>          | <b>94.98%</b>                | <b>39</b>   |
| <b>Gemini-2.5-pro</b> | 89.87%        | 86.60%                 | 89.12%                       | 37          |
| <b>Deepseek-r1</b>    | 89.42%        | 82.47%                 | 90.05%                       | 34          |
| <b>Qwen3-32B</b>      | 86.75%        | 79.38%                 | 85.02%                       | 30          |

**4.4.4 Model Sensitivity.** We evaluated four LLMs that support chain-of-thought reasoning to compare their effectiveness. The slight variances in performance reflect differences in the reasoning quality of the LLMs themselves. Across three runs per model on ACU, the average standard deviation for each model is approximately 1.78 percentage points; deterministic components such as RAG retrieval, behavior-tree validation, and rule-based compilation constrain the nondeterminism of the LLM-based stages. AUTOHIL demonstrates high adaptability, proving that effective HIL test generation is driven by our methodology rather than the specific choice of the LLM.

## 5 Related Work

**ECU System Test Generation from Requirements.** Effort are primarily focused on generating formal or semi-formal test cases from requirements. Gröpler et al. [20] constructed a spaCy-based NLP pipeline to convert requirements into an intermediate representation domain language. Khosrowjerdi et al. [28] utilized machine learning to generate tests from temporal logic requirements. Recently, LLMs have been adopted to handle unstructured requirements. Adabala et al. [3] utilized LLMs to generate test flows to reduce ambiguity in semi-formal requirements. Chow et al. [12] fine-tuned LLMs on domain-specific examples to capture the orchestration and naming conventions and generate test procedures. Milchevski et al. [36] transformed requirements into structured specifications. Xu et al. [44] used LLM to model requirements while SMT solvers to generate inputs that satisfy constraints. Zyberaj et al. [47] integrated LLMs with visual language models to transform requirements and system diagrams into structured test scripts. These approaches neglect integration with HIL benches. Our work considers constraints of HIL and generates test scripts on HIL benches.

**HIL-Based ECU System Functional Testing.** When performing HIL testing to validate ECU system functionality, existing research relies on platform-specific solutions that cannot be generalized, and only achieves limited automation. Formica et al. [17] applied search-based software testing in industrial simulations, using functional requirements and drivability metrics as oracles to parametrize test sequences, and finally ran on MARCdrive bench to validate SIL results. Karlsson et al. [26] integrated GitHub Copilot into Volvo’s test automation framework (TAF), utilizing platform-dependent C# templates to guide the generation of diagnostic test scripts. Wynn-Williams et al. [43] provided bench-specific context to LLMs, translating steps into instruction sequences of test bench (ecu.test toolchain). Feng et al. [16] proposed HIL-GPT for HIL scenarios within Vector



CAPL ecosystem, which employed fine-tuned embedding models to efficiently retrieve and utilize dispersed testing assets, yet focused on asset retrieval rather than test logic generation.

**Automated Test Generation for MCU and Embedded Systems.** Existing approaches largely prioritize robustness and low-level stability over functional correctness. EcuFuzz [11] incorporated a peripheral emulator to target SPI and CAN protocols, providing a novel fuzzing framework for the testing of hardware-dependent MCUs. Firmware re-hosting advancements enable testbed-free environments. FirmWire [21] utilized vendor-specific modeling to emulate cellular basebands, while FirmAE [30] introduced arbitrated emulation, to automatically detect boot failures caused by missing peripherals. These general-purpose methods lack the domain-specific context for ECU HIL testing.

## 6 Conclusion

We proposed AutoHIL, an end-to-end, LLM-empowered framework that automatically generates executable ECU system functional test scripts on HIL benches. Our evaluation demonstrates high effectiveness of test generation, and substantial efficiency gains over manual scripting.

## 7 Data Availability

The complete source code and experimental data of AutoHIL are available at our replication site [7]. However, the source code for the ECUs and the HIL bench is not included because it is proprietary to our industrial collaborator and protected by confidentiality agreements.

## Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 92582205).

## References

- [1] Mohammad Abboush, Christoph Knieke, and Andreas Rausch. 2024. A virtual testing framework for real-time validation of automotive software systems based on hardware in the loop and fault injection. *Sensors* 24, 12 (2024), 3733. doi:10.3390/s24123733
- [2] Mohammad Abboush, Christoph Knieke, and Andreas Rausch. 2025. Advancing real-time validation of automotive software systems via continuous integration and intelligent failure analysis. *Scientific Reports* 15, 1 (2025), 32936. doi:10.1038/s41598-025-21416-5
- [3] Bhargav Adabala, Gerhard Griessnig, Adam Schnellbach, Martin Ringdorfer, Christian Santer, Aisha Maria Puchleitner, Kaan Suar, Martin Mandl, and Vanesa Kloplic. 2024. AI-Driven Test Flow Generation from Semi-formal Functional Safety Requirements. In *Proceedings of the European Conference on Software Process Improvement*. 197–205. doi:10.1007/978-3-031-71139-8\_13
- [4] Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2024. Codemirage: Hallucinations in code generated by large language models. *arXiv preprint arXiv:2408.08333* (2024). doi:10.48550/arXiv.2408.08333
- [5] Ayman Amyan, Mohammad Abboush, Christoph Knieke, and Andreas Rausch. 2024. Automating Fault Test Cases Generation and Execution for Automotive Safety Validation via NLP and HIL Simulation. *Sensors* 24, 10 (2024), 3145. doi:10.3390/s24103145
- [6] Marina Araújo, Júlia Araújo, Romeu Oliveira, Lucas Romao, and Marcos Kalinowski. 2025. Domain knowledge in requirements engineering: A systematic mapping study. In *Proceedings of Euromicro Conference on Software Engineering and Advanced Applications*. 425–441. doi:10.1007/978-3-032-04200-2\_29
- [7] AutoHIL. 2026. AutoHIL-2026. <https://github.com/AUTOHIL/AutoHIL>
- [8] AUTOSAR. 2025. AUTOSAR Classic Platform. Accessed: 2025-12-13. <https://www.autosar.org/standards/classic-platform>
- [9] Mayank Bagari, Saurav Prajapati, Yunish Tuladhar, and Akshay Koti. 2025. *Use of LLM for Testcase Automation in Automotive Industry*. Technical Report. doi:10.4271/2025-28-0398
- [10] Arda Celik and Qusay H Mahmoud. 2025. A Review of Large Language Models for Automated Test Case Generation. *Machine Learning and Knowledge Extraction* 7, 3 (2025), 97. doi:10.3390/make7030097
- [11] Qicai Chen, Kun Hu, Sichen Gong, Bihuan Chen, Zikui Kong, Haowen Jiang, Bingkun Sun, You Lu, and Xin Peng. 2025. Structure-Aware, Diagnosis-Guided ECU Firmware Fuzzing. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 871–893. doi:10.1145/3728914

- [12] Man Yiu Chow and Mitsuhiro Kitani. 2025. Testing Procedure Generation Based on Testing Requirements for Automotive Components with LLM Fine-Tuning. In *Proceedings of International Conference on Applications of Natural Language to Information Systems*. 200–209. doi:10.1007/978-3-031-97144-0\_18
- [13] National Instruments Corporation. 2025. niveristand-python. Accessed: 2025-12-13. [https://niveristand-python.readthedocs.io/en/latest/api\\_reference.html](https://niveristand-python.readthedocs.io/en/latest/api_reference.html)
- [14] dSPACE GmbH. 2025. dSPACE. Accessed: 2025-12-13. <https://www.dspace.com/en/pub/home.cfm>
- [15] ASAM e.V. 2025. ASAM XIL. Accessed: 2025-12-13. <https://www.asam.net/standards/detail/xil/>
- [16] Chao Feng, Zihan Liu, Siddhant Gupta, Gongpei Cui, Jan von der Assen, and Burkhard Stiller. 2025. Smarter, not Bigger: Fine-Tuned RAG-Enhanced LLMs for Automotive HIL Testing. *arXiv preprint arXiv:2511.22584* (2025). doi:10.48550/arXiv.2511.22584
- [17] Federico Formica, Nicholas Petrunti, Lucas Bruck, Vera Pantelic, Mark Lawford, and Claudio Menghi. 2023. Test case generation for drivability requirements of an automotive cruise controller: An experience with an industrial simulator. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1949–1960. doi:10.1145/3611643.3613894
- [18] Ruizhi Gao, Jeong Soo Eo, W Eric Wong, Xinwei Gao, and Shou-Yu Lee. 2014. An empirical study of requirements-based test generation on an automobile control system. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1094–1099. doi:10.1145/2554850.2554934
- [19] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Swaib Dragule, and Andrzej Wąsowski. 2020. Behavior trees in action: a study of robotics applications. In *Proceedings of the 13th ACM SIGPLAN international conference on software language engineering*. 196–209. doi:10.1145/3426425.3426942
- [20] Robin Gröpler, Viju Sudhi, Emilio José Calleja García, and Andre Bergmann. 2021. NLP-Based Requirements Formalization for Automatic Test Case Generation.. In *Proceedings of CS&P*. 18–30.
- [21] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. 2022. FIRMWIRE: Transparent dynamic analysis for cellular baseband firmware. In *Network and Distributed Systems Security Symposium (NDSS)* 2022. doi:10.14722/ndss.2022.23136
- [22] Typhoon HIL Inc. 2025. HIL API — Typhoon API 2025.4 documentation. Accessed: 2025-12-13. [https://www.typhoon-hil.com/documentation/typhoon-hil-api-documentation/hil\\_api.html](https://www.typhoon-hil.com/documentation/typhoon-hil-api-documentation/hil_api.html)
- [23] International Organization for Standardization. 2011. *ISO 26262: Road Vehicles - Functional Safety*. Standard 26262. ISO.
- [24] Katharina Juhnke, Matthias Tichy, and Frank Houdek. 2021. Challenges concerning test case specifications in automotive software testing: assessment of frequency and criticality. *Software Quality Journal* 29, 1 (2021), 39–100. doi:10.1007/s11219-020-09523-0
- [25] Sunil Kamalakar, Stephen H Edwards, and Tung M Dao. 2013. Automatically generating tests from natural language descriptions of software behavior. In *Proceedings of the International Conference on Evaluation of Novel Software Approaches to Software Engineering*. 238–245. doi:10.5220/0004566002380245
- [26] Albin Karlsson, Erik Lindmaa, Simin Sun, and Mirosław Staron. 2024. AI-Based Automotive Test Case Generation: An Action Research Study on Integration of Generative AI into Test Automation Frameworks. In *Proceedings of the International Conference on Product-Focused Software Process Improvement*. 50–66. doi:10.1007/978-3-031-78392-0\_4
- [27] Rashidah Kasauli, Eric Knauss, Joyce Nakatumba-Nabende, and Benjamin Kanagwa. 2020. Agile islands in a waterfall environment: Challenges and strategies in automotive. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. 31–40. doi:10.1145/3383219.3383223
- [28] Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. 2017. Learning-based testing for safety critical automotive applications. In *Proceedings of International Symposium on Model-Based Safety and Assessment*. 197–211. doi:10.1007/978-3-319-64119-5\_13
- [29] Hojat Khosrowjerdi, Karl Meinke, and Andreas Rasmusson. 2018. Virtualized-fault injection testing: A machine learning approach. In *Proceedings of 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 297–308. doi:10.1109/ICST.2018.00037
- [30] Mingyun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. Firmac: Towards large-scale emulation of iot firmware for dynamic analysis. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 733–745. doi:10.1145/3427228.3427294
- [31] GP Kulk and Chris Verhoef. 2008. Quantifying requirements volatility effects. *Science of computer programming* 72, 3 (2008), 136–175. doi:10.1016/j.scico.2008.04.003
- [32] Peter A Lindsay, Sentot Kromodimoeljo, Paul A Strooper, and Mohamed Almorsy. 2015. Automation of test case generation from behavior tree requirements models. In *Proceedings of 2015 24th Australasian Software Engineering Conference*. 118–127. doi:10.1109/ASWEC.2015.23
- [33] Sidi Lu and Weisong Shi. 2023. Vehicle computing: Vision and challenges. *Journal of Information and Intelligence* 1, 1 (2023), 23–35. doi:10.1016/j.jiixd.2022.10.001
- [34] MathWorks. 2025. Simulink Test. Accessed: 2025-12-13. <https://www.mathworks.com/products/simulink-test.html>

- [35] Franc Mihalič, Mitja Truntič, and Alenka Hren. 2022. Hardware-in-the-loop simulations: A historical overview of engineering challenges. *Electronics* 11, 15 (2022), 2462. doi:10.3390/electronics11152462
- [36] Dragan Milchevski, Gordon Frank, Anna Hättö, Bingqing Wang, Xiaowei Zhou, and Zhe Feng. 2025. Multi-Step Generation of Test Specifications using Large Language Models for System-Level Requirements. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*. 132–146. doi:10.18653/v1/2025.acl-industry.11
- [37] Alexandros Mouzakitis, David Copp, Richard Parker, and Keith Burnham. 2009. Hardware-in-the-loop system for testing automotive ECU diagnostic software. *Measurement and control* 42, 8 (2009), 238–245. doi:10.1177/002029400904200803
- [38] Inc. OPAL-RT TECHNOLOGIES. 2025. RT-LAB Documentation. Accessed: 2025-12-13. <https://opal-rt.atlassian.net/wiki/spaces/PRD/overview>
- [39] Sven Peldszus, Noubar Akopian, and Thorsten Berger. 2023. RobotBT: Behavior-tree-based test-case specification for the robot framework. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1503–1506. doi:10.1145/3597926.3604924
- [40] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval augmentation reduces hallucination in conversation. *arXiv preprint arXiv:2104.07567* (2021). doi:10.48550/arXiv.2104.07567
- [41] Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. 2025. Automating a Complete Software Test Process Using LLMs: An Automotive Case Study. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 373–384. doi:10.1109/ICSE55347.2025.00211
- [42] Carsten Wiecher, Jannik Fischbadh, Joel Greenyer, Andreas Vogelsang, Carsten Wolff, and Roman Dumitrescu. 2021. Integrated and iterative requirements analysis and test specification: A case study at kostal. In *Proceedings of 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 112–122. doi:10.1109/MODELS50736.2021.00020
- [43] Stephen Wynn-Williams, Ryan Tyrrell, Vera Pantelic, Mark Lawford, Claudio Menghi, Phaneendra Nalla, and Hassan Artail. 2025. Can Generative AI Produce Test Cases? An Experience from the Automotive Domain. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 456–467. doi:10.1145/3696630.3728568
- [44] Tong Xu, Zheng Zhou, Xiaohong Chen, Zhiyi Xue, Yi Zhao, Min Zhang, and Zhi Jin. 2025. Requirements Dependency Driven Test Case Generation: An Automotive Industry Practice. In *Proceedings of 2025 IEEE 33rd International Requirements Engineering Conference (RE)*. 400–404. doi:10.1109/RE63999.2025.00046
- [45] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of 2014 IEEE symposium on security and privacy*. 590–604. doi:10.1109/SP.2014.44
- [46] Wang Zhang, Meng Shi, Xinglong Liu, and Linjie Ren. 2025. Proposing an Information Management Framework for Efficient Testing Strategies of Automotive Integration. *Processes* 13, 10 (2025), 3296. doi:10.3390/pr13103296
- [47] Denesa Zyberaj, Lukasz Mazur, Nenad Petrovic, Pankhuri Verma, Pascal Hirmer, Dirk Slama, Xiangwei Cheng, and Alois Knoll. 2025. GenAI-based test case generation and execution in SDV platform. *arXiv preprint arXiv:2509.05112* (2025). doi:10.48550/arXiv.2509.05112

Received 2026-01-30; accepted 2026-04-16