International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 3, May-June 2024||

DOI:10.15662/IJRPETM.2024.0703007

Domain-Oriented BDD using Pytest: A Design-Focused Approach to Reusable Behaviour Specifications

Chiranjeevulu Reddy Kasaram

Independent Researcher, USA

chiran.reddy16@gmail.com

ABSTRACT: Automation testing often suffers from brittle scripts that are difficult to maintain and poorly aligned with business requirements. Behavior-Driven Development (BDD) emerged to address this gap by enabling stakeholders to collaborate through a shared, executable specification language. However, without careful structural design, BDD implementations can quickly degrade into unreadable and fragile codebases. This paper proposes an approach that integrates Pytest-BDD with Domain-Oriented Test Design (DOTD) to create reusable, maintainable, and communication-rich behavioural specifications. By combining BDD's collaborative philosophy, the flexibility of the Pytest ecosystem, and the domain-driven structuring principles of DOTD, the framework ensures that test suites serve not only as verification tools but also as living documentation. This alignment fosters resilience to change, enhances stakeholder communication, and embeds test artefacts within the broader software lifecycle.

KEYWORDS: Pytest-BDD, BDD, domain-oriented design, test automation

I. INTRODUCTION

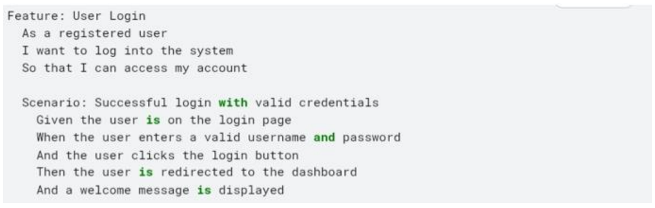
The current software testing environment is fraught with issues with the old-fashioned automation scripts often emerging as a maintenance nightmare with brittle behaviour, implementation, and telling you nothing about system behaviour [1]. This is its weakness because there is a fundamental lack of connection between technical implementation of tests and business requirements in which the tests are supposed to be validated. One of the paradigms that were established to solve this communication breakdown was Behavior-Driven Development (BDD) in a bid to establish cooperation between the technical and non-technical stakeholders through a common ubiquitous language in the form of executable specifications [2]. Despite the philosophical merits of BDD being well known, its practical implementation is fraught with pitfalls, unless the test suites are suitably designed, they degenerate into the same fragile and unreadable code that the test suites were supposed to be replacing [3]. The tooling that they are supporting, and the supporting structural design of the test codebase are highly significant in the efficacy of BDD [4]. In this paper, it will be proposed that a powerful way of creating reusable, maintainable and communication rich behavioral specifications is based on a synergetic integration of a powerful Python framework Pytest-BDD with Domain-Oriented Test Design (DOTD) concepts. This approach relies on performing more than the required verification of code to create living documentation that defines and demonstrates system behavior at a domain level, so that the tests are resistant to change, and in fact of benefit to the entire software lifecycle [1], [5].

II. BACKGROUND AND KEY CONCEPTS

The methodology would be approach-oriented towards a combination of three pillars of support, which would be; collaborative philosophy of the Behavior-Driven Development, the powerful tooling of the Pytest ecosystem and the structural principles of Domain-Oriented Test Design. One should study how each of them works separately, in order to enjoy the harmony of the functions of each.

Behavior-Driven Development (BDD) & Gherkin Behavior-Driven.

Agile Software development process is development software development that encourages collaboration among the developers, quality assurance experts and non-technical business stakeholders. Test-Driven Development (TDD), in which the focus is not on code functionality testing, is an expansion of it, where desired system behavior is defined and validated, in the eyes of the stakeholder [1]. The Gherkin language, providing a natural syntax of executable specifications expressed in a structured form (mostly Given-When-Then), is the most important in this practice. The syntax constitutes a ubiquitous language, which is common code that is known to everyone, and this minimizes the chances of misinterpreting the requirements [2]. These specifications are papers in the form of feature files which are live documentation and acceptance criteria.



```
Feature: User Login
  As a registered user
  I want to log into the system
  So that I can access my account

  Scenario: Successful login with valid credentials
    Given the user is on the login page
    When the user enters a valid username and password
    And the user clicks the login button
    Then the user is redirected to the dashboard
    And a welcome message is displayed
```

Figure 1. Example Gherkin .feature file for a login feature

Pytest and the Pytest-BDD Plugin

Pytest is now one of the most popular Python tests frameworks with a simple syntax, powerful setup and teardown infrastructure in fixtures, and a rich ecosystem of Python test and debugging tools [4]. These strong points are taken advantage of in the Pytest-BDD plug-in since it uses the BDD practices. Unlike other independent systems like Cucumber, Pytest-BDD is designed to integrate into the current workflow of Pytest framework and Testers writing their step definitions with traditional assertions, parametrization and injecting dependency in their step definitions [3]. This integration is a strength as other frameworks can frequently require context switching and recreation of toolchains, and BDD adoption can therefore be more easily accessible to Python based projects.

Domain-Oriented Test Design (DOTD)

Domain-Oriented Test Design is a strategic approach to structuring the test code around the primary business concepts and business capabilities of the business being tested, instead of the implementation details of the application like user interface components, or API endpoints [5]. The central concept of DOTD is that test logic must be written in the vocabulary of the domain (e.g. customer, account, login) and that it should be without concern with how the interface of the application is written. This is typically achieved by creating an abstraction layer—often composed of page objects, service clients, or other domain entities—that encapsulates all interactions with the system. Tests and step definitions then orchestrate behavior through this domain layer, decoupling the specification from the implementation.

TABLE I. CONCEPTUAL DIAGRAM CONTRASTING UI-ORIENTED AND DOMAIN-ORIENTED TEST STRUCTURES

Feature	UI-Oriented Test Structure	Domain-Oriented Test Structure
Hierarchy	Flat and tightly coupled to the user interface.	Layered and clearly separated, with tests focused at each level.
Focus	How the user interacts with the application.	The core business logic and behavior.
Test types	Primarily end-to-end tests that drive the UI.	A pyramid of tests: many fast unit tests, some component tests, and a few slow UI tests.
Coupling	High coupling; tests are fragile and break easily when the UI changes.	Low coupling; the business logic is tested independently of the UI.
Speed	Slow to run, as they involve the full application stack.	Fast, with the majority of tests running quickly in isolation.
Debugging	Difficult to isolate the root cause of a	Failures are localized to the specific unit or

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 3, May-June 2024||

DOI:10.15662/IJRPETM.2024.0703007

	failure.	component being tested.
Language	Uses terms related to UI elements (e.g., "click button X").	Uses the "ubiquitous language" of the business domain.

The Problem: Brittleness in Implementation-Coupled Tests

Another nearby anti-pattern is that the theoretical advantages of BDD are frequently wrecked in practice by a simple answer to the question of how to define Gherkin step definitions: direct dependence on the volatile implementation details of the user interface of the application. Another common, but flawed, strategy is writing step definitions which are simple, scripted automations of the UI. An example of this is a step where a direct CSS selector is used, and it is written as When I enter test user in the element hash username. This method mixes the behavioural specification (the what) with the technical implementation (the how) and results in a number of critical weaknesses [1], [4].

The consequences of this interconnection are fatal. First, it makes it too fragile; any minor change that a programmer will do to the UI such as adding an HTML id attribute to one of their HTML elements in order to polish it will make all the tests dependent on it fail instantly and the programmer will have to spend time hunting down the entire codebase and do some changes in it, which are difficult to know [8]. Second, it causes atrocious reusability. Step is so narrow to a specific UI object on one page that reuse is not possible in other contexts and functionality, leading to gigantic code multiplication. Thirdly, it derails the communicative intention of BDD. The Gherkin situations are lost in technicalities and are no use as a language between the business stakeholders or they are in essence a doujin language and there are two parallel specifications: the feature file and the one hiding in the step definition code [2], [5].

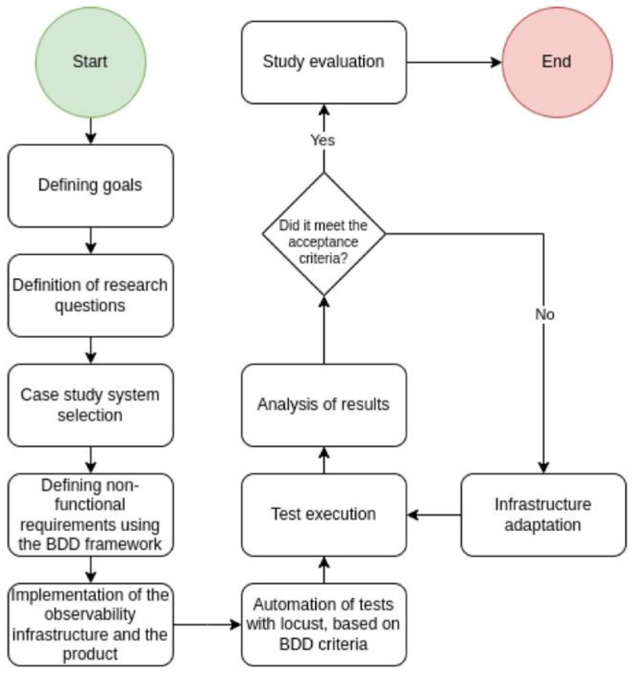

Start
Defining goals
Definition of research questions
Case study system selection
Defining non-functional requirements using the BDD framework
Implementation of the observability infrastructure and the product
Automation of tests with locust, based on BDD criteria
Test execution
Analysis of results
Did it meet the acceptance criteria?
Yes
No
Infrastructure adaptation
Study evaluation
End

Figure 2. The Fragile Automation Anti-Pattern

III. PROPOSED METHODOLOGY: INTEGRATING PYTEST-BDD WITH DOTD

The implementation-coupled tests themselves are very weak, which makes it necessary to have a good architectural solution. This is what the proposed methodology focuses on, and it imposes a strict separation of concerns and leverages the capabilities of Pytest-BDD to implement a Domain-Oriented Test Design. This leads to a multi-layered test structure where each of the layers is endowed with a distinct responsibility in order to promote a sense of reusability and maintainability along with clarity.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 3, May-June 2024||

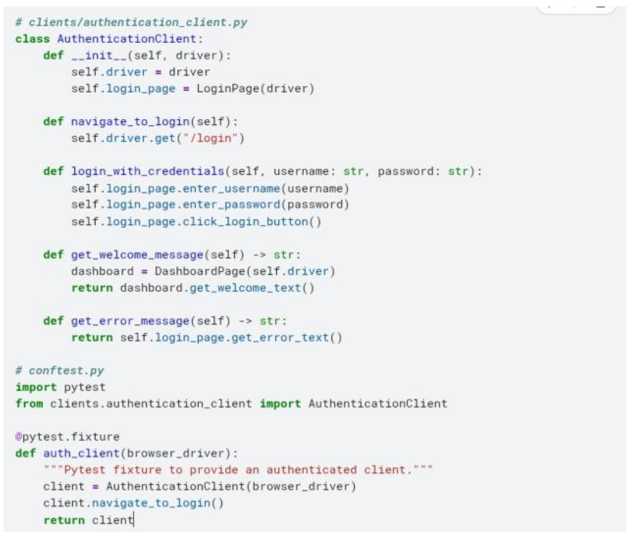
DOI:10.15662/IJRPETM.2024.0703007

Philosophical Underpinning:
Separation of Concerns The crux of this principle is that the definition of the behavior should be strictly separated from the mechanism of implementation. The Gherkin feature files are to be maintained clean with the specifications of what the system should do in the clear business language. These step definitions are translators that identify the action in the natural language and pass it on to a domain specific layer. This domain layer, again, carries the information about the interface with a system, be it a web interface or an API or database. This abstraction ensures that the changes in the implementation details are contained in a single layer and hence the higher-level specification are not exposed to change [1], [4].

The Domain Layer:
Reusable Fixtures and Services This pattern consists in introducing a domain layer, which is created with Pytest fixtures and client classes. This layer is the source of a stable application-specific API that testing can depend on. Tests communicate with real domain objects such as user, account, or dashboard instead of having to deal with buttons and fields.

A Pytest fixture is configured to make a shared domain client instance available to the definitions of the steps. The interactions of a certain domain area are contained in this client class.

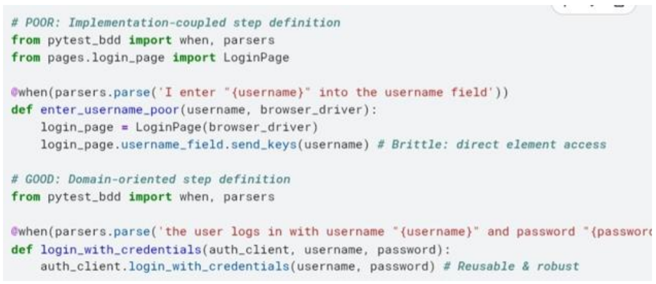
Code Snippet 1: Example of a domain-layer AuthenticationClient class.



```python
# clients/authentication_client.py
class AuthenticationClient:
    def __init__(self, driver):
        self.driver = driver
        self.login_page = LoginPage(driver)

    def navigate_to_login(self):
        self.driver.get("/login")

    def login_with_credentials(self, username: str, password: str):
        self.login_page.enter_username(username)
        self.login_page.enter_password(password)
        self.login_page.click_login_button()

    def get_welcome_message(self) -> str:
        dashboard = DashboardPage(self.driver)
        return dashboard.get_welcome_text()

    def get_error_message(self) -> str:
        return self.login_page.get_error_text()

# conftest.py
import pytest
from clients.authentication_client import AuthenticationClient

@pytest.fixture
def auth_client(browser_driver):
    """Pytest fixture to provide an authenticated client."""
    client = AuthenticationClient(browser_driver)
    client.navigate_to_login()
    return client
```

Structuring Step Definitions
Structuring Step Definitions for Reusability With the domain layer in place, step definitions become thin, declarative orchestrators. Their sole purpose is to extract parameters from the Gherkin steps and call the appropriate method on the injected domain client. This eliminates all implementation details from the step code.

Code Snippet 2: Contrasting a coupled vs. A decoupled step definition



```python
# POOR: Implementation-coupled step definition
from pytest_bdd import when, parsers
from pages.login_page import LoginPage

@when(parsers.parse('I enter "{username}" into the username field'))
def enter_username_poor(username, browser_driver):
    login_page = LoginPage(browser_driver)
    login_page.username_field.send_keys(username) # Brittle: direct element access

# GOOD: Domain-oriented step definition
from pytest_bdd import when, parsers

@when(parsers.parse('the user logs in with username "{username}" and password "{passwor
def login_with_credentials(auth_client, username, password):
    auth_client.login_with_credentials(username, password) # Reusable & robust
```

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 3, May-June 2024||

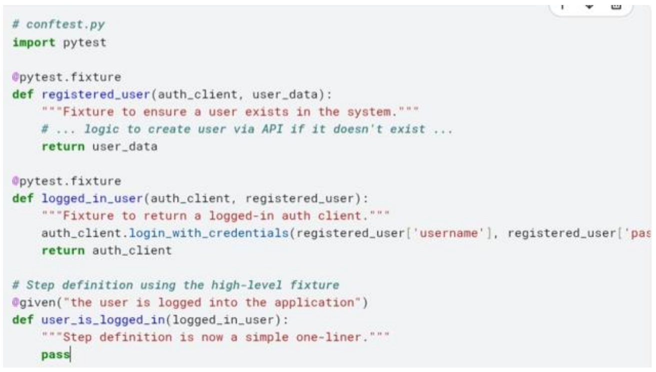
DOI:10.15662/IJRPETM.2024.0703007

The step definition of GOOD knows nothing of whether login with credentials is done through a web UI or a REST API request or a child command. This renders it very reusable and strong.

The Power of Pytest Fixtures Dependency Injection

The strength of Pytest Fixtures The fixtures system in Pytest is what glues this architecture together, and it performs the dependency injection automatically providing a definition of the steps with the actual domain clients they need. They can be scoped (e.g. function, session) to configure them efficiently and can contain other fixtures, which enables them to be a complex construction, such as authenticate a user a user, to be written once and reused in all.

Code Snippet 3: Using fixture composition for complex setup.



```python
# conftest.py
import pytest

@pytest.fixture
def registered_user(auth_client, user_data):
    """Fixture to ensure a user exists in the system."""
    # ... logic to create user via API if it doesn't exist ...
    return user_data

@pytest.fixture
def logged_in_user(auth_client, registered_user):
    """Fixture to return a logged-in auth client."""
    auth_client.login_with_credentials(registered_user['username'], registered_user['pas
    return auth_client

# Step definition using the high-level fixture
@given("the user is logged into the application")
def user_is_logged_in(logged_in_user):
    """Step definition is now a simple one-liner."""
    pass
```

Whole Process

The general architecture and the entire process are a clean, unidirectional flow which has no concerns and has minimal couplings. Test implementation is executed using correct route and hence the implementation does not define the behavior.

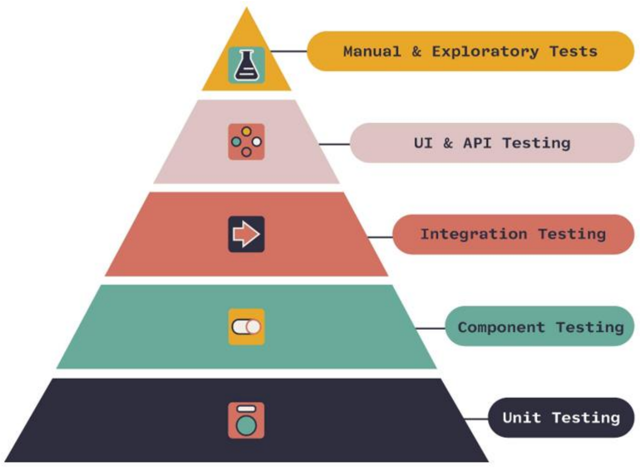


Figure 3. Flowchart of the Domain-Oriented Pytest-BDD Test Architecture

IV. CASE STUDY: IMPLEMENTING A USER AUTHENTICATION FLOW

In order to empirically confirm the practical usefulness of combining Pytest-BDD with Domain-Oriented Test Design, a thorough analysis of a user authentication system can be used as a perfect case study. This situation, which includes successful and failed login, and account lockdown, is an excellent illustration of how a traditional implementation-based approach is weak and how the proposed approach is robust.

It begins with the Gherkin language that unambiguously defines desired behaviors. Figure 5 illustrates the three user journeys that are featured in the scope of the feature specification, and they are successful authentication, invalid credentials, and handling security procedures in the event that there are multiple failures. More to the point, this specification is not beyond the scope of the problem, informing you of the user purpose and system output without a single button, field, or other details of the implementation. This conforms to the BDD notion of establishing a single source of truth that can be comprehended by all the stakeholders [1], [2].

The essential aspect of domain-oriented approach is the development of powerful Authentication Client. This component is believed to be the sole entry point between the tests and system under test and the business logic of what actions can be undertaken in relation to authentication. Instead of possessing some piece of broken and fragmented code to work with UI elements, this client has sensible and comprehensible ways, such as logging in with legitimate credentials and trying to log in more than once. The details of the realization of how these actions will be implemented, whether using a web form, an API call or any other schemes, are completely represented in this layer. The brittleness observed by Irshad et al. [8] directly has a solution in this design as the effect of variations to the structure of the UI is absorbed by the client.

The step definitions of this architecture are considerably transformed. They lose their former role as detailed automation scripts and get to be slim orchestrators. They are left with the task of de-serializing the parameters of the Gherkin steps and sending it out to the domain client being injected with whatever method is needed to execute it. As an example, the logging in step just invokes the appropriate client method and provides the credentials. This prevents code duplication and also produces very reusable steps because the same log in step can be re-used on many feature files without alteration, which is much better than the fixed situation described by Mughal [2].

The final trial of this design is how it reacts to change. As an example, a typical change would be the developer changes the HTML attribute which contains the text of an error message on the login page. In a conventional, coupled test suite, this would require a tiresome and error-prone search through all the step definitions that were directly using the old selector and could cause dozens of tests to fail. Under the domain-oriented architecture proposed, the method that encapsulates the business requirements is the get-error-message method of the Authentication Client class, which is the only point that is changed. The step definitions that verify error messages, which depend on this method, remain entirely untouched. The topmost abstraction of feature are the Gherkin feature files which are totally immutable. This proves that there is a radical cut in the overhead maintenance costs, and it confirms the fact that the model can make test suites that are not only specifications executable but also long-lasting assets that are not affected by the unavoidable changes of the application [4], [8]. This case study shows conclusively that the combination of Pytest-BDD and Domain-Oriented Test Design are effective in mitigating the fundamental issues of conventional test automation.

V. EVALUATION AND DISCUSSION

The implemented case study reveals that implementation of Pytest-BDD, and Domain-Oriented Test Design introduce tremendous and tangible value. This advantage is largely a theatrical boost in sustainability. The implementation details being abstracted in a domain layer mean that application changes in the UI or API can be made at a single point, greatly reducing the cost and effort of maintaining a test suite compared to more traditional coupled designs [4, 8]. This is directly proportional to greater robustness, because the specifications of the tests will not be subjected to the volatile implementation details of the tested system, and the test suite itself will be a more stable and reliable one.

Moreover, better reusability is developed by this methodology. The step definitions and domain client functions become composable building blocks that can easily be joined together to form new test scenarios without code duplication [1, 2]. The method is also effective to maintain the communicative value of BDD. The remaining features of the Gherkin are clean, declarative and user behavioral and hence useful as a living documentation to both developers, testers and business stakeholders [5]. Although the initial cost due to the domain layer might be increased, the cost-saving in maintenance and the improvement of the clarity of the test suite in the long-term perspective result in a strong return on investment, which practically resolves the fundamental drawbacks of implementation-coupled test automation.

VI. CONCLUSION AND FUTURE WORK

As has been shown in this essay, the Pytest-BDD framework with Domain-Oriented Test Design principles is a strong and efficient methodology of building quality test suites. This strategy directly addresses the brittle nature of the traditional test automation difficulty of maintainability and low reusability by ensuring that feature files specify the behavior, step definitions make calls to domain objects, and an explicit client layer is created to support the

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 7, Issue 3, May-June 2024||

DOI:10.15662/IJRPETM.2024.0703007

implementation. The resulting test suite is reusable behavior specifications and serves as both reliable validation mechanism and documentation that is readable by all interested parties [1], [4].

Attempts to automatically generate domain-layer client code by processing API specification in formats like the OpenAIP will be sought in the future and could accelerate the initial setup of this architecture. Further developed patterns of the management of complex state and data set up across complex domain workflows, and the seamless integration of non-functional testing of such workflows, such as performance checks, into this domain oriented BDD structure should also be explored further.

REFERENCES

[1] J. F. Smart and J. Molak, BDD in Action: Behavior-driven development for the whole software lifecycle. Simon and Schuster, 2023.
[2] A. H. Mughal, “Advancing BDD Software Testing: Dynamic Scenario Re-Usability And Step Auto-Complete For Cucumber Framework,” arXiv preprint arXiv:2402.15928, 2024.
[3] J. Lång, “In-depth comparison of BDD testing frameworks for Java,” 2022.
[4] A. Molina, Crafting Test-Driven Software with Python: Write test suites that scale with your applications' needs and complexity using Python and PyTest. Packt Publishing Ltd., 2021.
[5] T. Zameni, P. van Den Bos, J. Tretmans, J. Foederer, and A. Rensink, “From BDD scenarios to test case generation,” in Proc. 2023 IEEE Int. Conf. Softw. Testing, Verification and Validation Workshops (ICSTW), Apr. 2023, pp. 36–44.
[6] R. Chanin, A. Santos, and N. Nascimento, “Teaching BDD in Active Learning Environments: A Multi-study Analysis,” in Proc. 14th Int. Conf. Computer Supported Education (CSEDU), Brasil, 2022.
[7] N. P. D. Nascimento, “A study of teaching BDD in active learning environments,” 2020.
[8] M. Irshad, J. Börstler, and K. Petersen, “Supporting refactoring of BDD specifications—An empirical study,” Information and Software Technology, vol. 141, p. 106717, 2022.
[9] V. Mrazek, “Optimization of BDD-based approximation error metrics calculations,” in Proc. 2022 IEEE Computer Society Annu. Symp. VLSI (ISVLSI), Jul. 2022, pp. 86–91.