Test-Driven Generation of Behavioral Threads from Specifications: An Industrial Experience Report

Giovanni Rosa*, David Moreno-Lumbreras*, Gregorio Robles* and Jesús M. González Barahona*

Universidad Rey Juan Carlos, Spain

Abstract

Scenario-based Programming (SBP) is a paradigm for reactive systems that lets developers specify functionality as independent, interacting scenarios called behavioral threads (b-threads). Despite its modularity benefits, BP adoption in industry is limited by the expertise required to design and implement b-threads and the effort required to define the behavioral scenarios. We present a TDD-driven code generation workflow that generates b-threads directly from natural-language behavior specifications, using a generate-test-fix loop to ensure reliability and adherence to the requirements. We evaluate the approach in an industrial case study, measuring (i) the functional correctness via the $pass@k$ metric across five different behaviors, and (ii) practitioner perception through semi-structured interviews. The top models (gemma-4-26b-a4b-it and GPT-OSS-120B) achieve 100% pass rate, and the initial feedback from two interviewed practitioners indicates high trust in the approach due to the plan-code-test cycle. While preliminary, these results are promising but highlight the need for further work to get a more reliable and generalizable solution. We discuss the implications and takeaways, as well as the limitations for the adoption in practice.

Keywords

Scenario-Based Programming, Code Generation, Large Language Models, Empirical Software Engineering

1. Introduction

Test-Driven Development (TDD) [1] is a software development methodology that focuses more on the definition of test cases, right after the functional requirements, before implementing the actual code. TDD has been evaluated in the software industry, improving the quality of the code by reducing the number of defects and increasing the maintainability of the code [2]. A modern shift in the software development industry is the adoption of Large Language Models (LLMs) for code generation, which has been shown to be effective when a proper guidance is provided, such as Test-driven code generation [3, 4, 5, 6], where the LLMs are guided by the test cases defined for the input task, and then they are used to verify the correctness of the generated code.

Scenario-Based Programming (SBP) [7] is a programming paradigm allowing developers to design and implement complex and modular systems by defining independent scenarios, namely *behavioral threads* (*b-threads*), which encapsulate the event flows of a specific behavior. However, adopting SBP is not trivial, as it requires a proper understanding of the behavioral programming paradigm in order to design and define complex behaviors.

The work reported in this paper is part of the research activities conducted within ADVISE, a research project that is a collaboration between the Universidad Rey Juan Carlos and the Osoco company[1]. In particular, Osoco is adopting scenario-based programming for some of their internal software products, and they are interested in evaluating the automated generation of behavioral threads from a set of behavioral requirements, in order to reduce the effort required to define the behaviors and to facilitate

Woodstock'22: Symposium on the irreproducible science, June 07–11, 2022, Woodstock, NY

*Corresponding authors.

giovanni.rosa@urjc.es (G. Rosa); david.morenolu@urjc.es (D. Moreno-Lumbreras); grex@gsyc.urjc.es (G. Robles); jesus.gonzalez.barahona@urjc.es (J. M. G. Barahona)

0000-0002-5241-1608 (G. Rosa); 0000-0002-5454-7808 (D. Moreno-Lumbreras); 0000-0002-1442-6761 (G. Robles); 0000-0001-9682-460X (J. M. G. Barahona)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1] https://osoco.es/

the adoption of SBP in their development process. A possible solution is to leverage the concept of Test-Driven Code Generation, where LLMs are guided by the behavior descriptions and using event flows as test cases, abstracting away the complexity of designing and implementing the behavioral threads.

In this paper, we aim to investigate this by evaluating the feasibility of using such an approach to generate a *b-threads* starting from an input specification, and assess the reliability of the generated code in terms of functional correctness and qualitative feedback from the developers from Osoco. The work is shaped as a preliminary study to evaluate the feasibility of the approach for generating a single *b-thread*, to allow a successive step focused on the management of multiple *b-threads*.

2. Related Work

The scientific literature investigated the effectiveness of TDD in improving the reliability of the code generated by LLMs. Piya et al. [5, 6] showed that combining TDD practices with LLMs improves the correctness of the solutions generated for Leetcode problems. Mathews et al. [4] proposed *TGen*, a framework for generating code solutions starting from a set of input test cases, integrating a remediation loop to iteratively improve the solution until the tests pass. Fakhoury et al. [3] proposed *TiCoder*, a workflow for interactive test-driven code generation, where LLMs generate tests and code solutions from input user requirements, with an iterative human-in-the-loop refinement. They both showed the improvements provided by their approaches on the *MBPP* [8] and *HumanEval* [9] datasets. We believe that, by adopting Test-Driven Code Generation for SBP, using test cases to formally describe the expected behavior and scenarios, it helps to reduce the complexity of implementing behavioral threads, automatically generating the code and ensuring its correctness.

Studies from the scientific literature already investigated the usage of LLMs for supporting the generation of code specific for Scenario-based programming. Harel et al. [10] explored the use of generative AI to augment scenario-based modeling, outlining a model and language-agnostic methodology combining LLMs output with the manual and automatic inspection of the resulting scenario-based models. Yaacov et al. [11] proposed an empirical evaluation for using LLMs to translate the natural language requirements in behavioral specifications (i.e., behavioral threads) using the *BPpy* [12] library. Berzack et al. [13] explored the usage of LLMs to support the generation of behavioral models via scenario-based programming, using also the *BPpy* library, providing a working agent implementation for the Connect4 game. None of the previous works investigated the adoption of an automated pipeline, integrating a self-fixing loop to ensure the correctness of the generated code, which is the open challenge that we aim to start exploring with our work putting the focus on the industrial adoption.

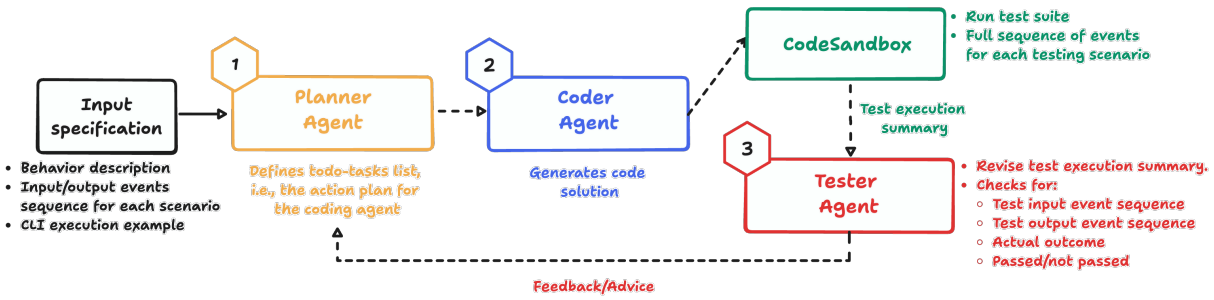


Figure 1: The TDD-driven code generation pipeline for behavioral programming.

3. TDD-for-BP Code Generation Pipeline

The proposed approach is based on the pipeline defined by Fakhoury et al. [3], which consists of three main phases: (i) code generation, having as input the problem specification and the test cases, (ii)

execute the generated code against the provided test cases, and (iii) providing back with feedback to the LLMs based on the test results, to reiterate the code generation process until the generated code passes all the test cases. In our case, the pipeline is adapted to the context of behavioral programming, where instead of the function, we have independent behavioral threads. The test cases are defined as input/output event sequences.

Figure 1 reports the proposed pipeline, consisting of three main agent entities. The **planner agent** is responsible for defining a set of actionable steps to provide directions to the code generation phase, by breaking down a big task into smaller, manageable steps. The **coder agent** is basically an executor, responsible for generating and revising the code implementation of the behavioral thread, based on the directives received from the planning agent. The **tester agent** has the scope of evaluating the test suite execution summary, check which test has failed, reason on them and provide back with feedback summarized from the test results to the planner agent, which will use this feedback to reiterate the behavioral thread generation process.

The pipeline starts with the planning phase, in which the planner analyzes the input specification and summarizes the implementation directives. Next, the coder agent generates the behavioral thread implementation, which is then executed against the test suite in an isolated code sandbox. The tester agent analyzes the test suite execution summary, providing fix advice back to the planner agent, which uses this feedback to reiterate the behavioral thread generation process. The generate-and-fix-loop continues until the generated behavioral thread passes all the test cases defined in the specification, or until a maximum number of iterations is reached (i.e., five).

The approach is designed to be flexible and not language-specific. The only constraint is the code sandbox, that must be defined based on the target programming language. For the purpose of this study, we focus on Python and the *BPpy* [12] library for behavioral programming.

4. Preliminary Study Design

The *goal* of the study is to evaluate whether the proposed Test-Driven Code Generation pipeline can generate correct and practically useful *b-threads* from the input behavior specifications. The study is driven by two main research questions:

- ***RQ1:*** *To what extent does the agentic workflow generate functionally correct b-threads from the input behavior specifications?* We aim to evaluate the effectiveness of the proposed agentic workflow measuring how often it is able to produce functionally correct code from a given specification.
- ***RQ2:*** *How do the industry practitioners perceive the agentic workflow for real development work?* We aim to assess qualitatively the perception of the industry practitioners on the generated b-threads and the applicability of the proposed agentic workflow in real development work.

4.1. Study Context

The context of the study is composed of a set of behavior specifications defined together with Osoco. Specifically, the specifications are reported in a GitHub Wiki[2], containing the overall description of the behavior and one or more example scenarios, containing the sequence of input/output events reporting the expected behavior of the system. The specifications are defined in a natural language, and they are then parsed and converted into a set of JSON files containing the specification converted in Markdown format and a list of input/output test cases, derived from the example scenarios defined in the specification. The JSON files are then used as input for the proposed agentic workflow, which generates the corresponding b-threads. The final dataset is composed of 5 behavior specifications for the Tic-tac-toe game.

[2]https://github.com/isegorg/bp-tictactoe-example/wiki

4.2. Experimental Procedure

In this section we describe the experimental procedure to answer the two research questions that drive the study.

4.2.1. RQ1. B-thread code-generation effectiveness

To answer the first RQ, we implemented and executed the proposed agentic workflow on the set of behavior specifications defined in the study context. The programming language used for the generated code is Python, using the *BPpy*, a library for behavioral programming in Python. We used Python and the *LangGraph* library to implement the single agents, and *Litellm* for the LLM inference.We implemented a custom code sandbox to execute the generated *b-threads*, consisting of a *FastApi* server taking as input the code implementation and the test cases input. To simplify the test suite execution and keep the pipeline implementation adaptable to different languages, we execute and evaluate the test cases using the stdin and stdout streams, where the input events are provided to the stdin stream, and the output events are read from the stdout stream. This means that the code generated by the models should also contain the harness to read the input events and write the output events.

The models are prompted with a mix of multi-agent and chain-of-tought (plan-before-coding), with few-shot and in-context examples to guide the generation of the code providing an API reference for the *Bppy* library. This is because in the first prototype, the model was not able to correctly use the library directives.

LLM Experimental Parameters. We fixed model *temperature=0.2* and *max_tokens=3000*, that are common parameters used for code-related tasks [14, 15], to reduce the non-determinism, but we give a larger threshold for the output to avoid truncating long outputs provided by the tester and planner agents, as they can contain longer explanations on how to fix the code and how to plan the next steps for the code generation. The models involved are some of the top-performing LLMs for code generation [16]. We prioritized open-weight models, to ensure the reproducibility of the experiments, and used *OpenRouter*[3] as inference API. We used: (i) *GPT-OSS* [17], the open-weight alternative for the GPT models, *20B* and *120B* variants; (ii) *Llama 4* [18] model family, in the *Maveric* and *Scout* variants, (iii) *Qwen3-Coder* [19] model family, in *30B* and *480B* variants, and (iv) *Gemma 4*, in its *26B* variant. The prompts and code harness used to execute the overall pipeline are reported in the replication package [20], to allow reproducibility of the experiments.

Evaluation Metrics. To evaluate the functional correctness of the generated code, and the adherence to the input behavior specification, we compute the *pass@k* metric [9]. A task is counted as solved if at least one of the top k generated candidates passes all tests. We execute a total of 5 pipeline executions of each behavior, and we compute the *pass@k* metric for *k=1,3,5.* A full pipeline execution is defined as a single iteration of the generate-test-fix loop, where the planner, coder and tester agents are executed in sequence. The pipeline is executed until the generated code passes all the test cases or until a maximum number of iterations is reached. We limit the maximum attempts of the generate-test-fix loop to five: This means that if the generated code does not pass all the test cases after five iterations, the iteration is stopped and the task is counted as unsolved. Additionally, we report the average number of iterations needed until success (or until the maximum number of iterations is reached), and the average number of tokens used for a single iteration, including the tokens used for the prompts and the tokens used for the responses of each agent.

4.2.2. RQ2. Practitioner assessment via structured interviews

We conduct a semi-structured interview with the developers of Osoco, to collect their feedback on the approach and the generated code. The research protocol is inspired by the guidelines for conducting semi-structured interviews in the context of Software engineering [21].

[3] https://openrouter.ai/models

The interview is composed of five interaction phases: An introductory part, where we explain the context of the study and the proposed agentic workflow, along with the structure of the interview. We ask some demographic questions such as experience with behavioral programming (phases 1), followed by questions about the perception of the approach, after showing the workflow (phases 2). Next, in phase 3, we present some of the generated solutions, and we ask the participants to assess, on a scale 1-5, the perceived quality of the generated code in terms of correctness, comprehension and maintainability. We selected the code generated for the *Square Taken* behavior, one of the most challenging behaviors, and we present the code generated by the two best-performing models, *gemma-4-26b-a4b-it* and *GPT-OSS-120B*, to have a common reference and a cross-check for the discussion. In phase four follows a set of short open-ended questions to collect the opinions on the applicability of the workflow in real development work, and on possible improvements on the input specification. All the questions are reported in the replication package [20], due to the space constraints, along with the interview guide used to conduct the interview. We close the interview with the cool-off (phase five), where we ask the participants to provide any additional comments or feedback that are not covered by the previous questions, and we provide the closing remarks on the interview and the study. We also leveraged the cool-off phase to ask the participants to cross-evaluate the code generated by the two models, to have a more objective assessment of the generated code. The total interview duration is $\sim$40 minutes. The interview is conducted by one of the authors having the technical expertise on the implementation of the proposed agentic workflow in order to answer any questions or doubts that may arise during the interview. The interview is conducted via conference call, in English, recorded and then transcribed for the successive analysis. This mitigates the risk of losing information during the interview, and allows focusing on the discussion with the participants. We ran two separate sessions, one for participant *P1* and one for participant *P2*, to avoid any influence of the participants on each other. The data analysis is conducted by two of the authors, from now on annotators, identifying common themes and patterns in the responses of the participants. The two annotators independently revised the coded tags and cross-validated the results where any discrepancies are resolved. For some of the Likert questions, the participants did not provide a specific score number, and we preferred to not interrupt the flow of the interview. Thus, we used a rubric to encode the score based on the participant's response, reported in the replication package [20]. The rubric and the final score have been cross-validated and discussed by the two annotators to ensure the reliability of the coding process. The final results are qualitatively discussed and reported in the following sections.

5. Experimental Results

In this section we report the results for the two research questions of our industrial experience report.

5.0.1. RQ1. B-thread code-generation effectiveness

Table 1 provides the execution details for each behavior. Here we can see that there are some behaviors that are more challenging than others, such as *Square Taken* and *Game Over*, which are not solved by some models even for $pass@5$. The average number of iterations and tokens used also varies across behaviors, with a higher number in the two cited behaviors.

gemma-4-26b-a4b-it and *GPT-OSS-120B* achieve the maximum $pass@k$ scores (100%) across all behaviors and k values, with the second model having a lower number of iterations and tokens used, indicating a more efficient generation process. On the other hand, *Llama-4-Maverick* and *Llama-4-Scout* are the least effective models, even with pass@5 they are not able to generate correct code for all behaviors.

An interesting observation is that *GPT-OSS-20B*, despite being a smaller model, is able to achieve 100% $pass@k$ for some behaviors, indicating that model size is not the only factor affecting performance. This means that smaller models can be effective with a tradeoff in terms of overall performance, suggesting that the choice of model should consider both performance and resource constraints.

Table 1
Detailed execution results for each behavior, including pass@k (k = 5), mean iterations ($\bar{I}$), and mean tokens per iteration ($\bar{T}$).

		pass@k (%)				
Model	**Behavior**	**@1**	**@3**	**@5**	$\bar{I}$	$\bar{T}$
gemma-4-26b-a4b-it	Alt. Turns	100	100	100	1 ± 0	$9\,529 \pm 126$
	Draw	100	100	100	2 ± 0	$16\,011 \pm 1\,733$
	Game Over	100	100	100	3 ± 1	$28\,983 \pm 1\,306$
	Player Wins	100	100	100	2 ± 1	$23\,775 \pm 2\,684$
	Square Taken	100	100	100	2 ± 1	$12\,861 \pm 1\,609$
gemma-4-26b-a4b-it	**Aggregate**	**100**	**100**	**100**	$\mathbf{2 \pm 1}$	$\mathbf{20\,551 \pm 7\,345}$
GPT-OSS-120B	Alt. Turns	100	100	100	1 ± 0	$10\,876 \pm 428$
	Draw	100	100	100	1 ± 0	$13\,681 \pm 672$
	Game Over	100	100	100	1 ± 0	$21\,616 \pm 1\,064$
	Player Wins	100	100	100	1 ± 1	$20\,693 \pm 2\,397$
	Square Taken	100	100	100	2 ± 0	$12\,658 \pm 1\,553$
GPT-OSS-120B	**Aggregate**	**100**	**100**	**100**	$\mathbf{1 \pm 1}$	$\mathbf{15\,641 \pm 4\,488}$
GPT-OSS-20B	Alt. Turns	100	100	100	2 ± 1	$11\,680 \pm 2\,278$
	Draw	80	100	100	3 ± 2	$17\,734 \pm 3\,685$
	Game Over	80	100	100	3 ± 1	$23\,176 \pm 1\,580$
	Player Wins	60	100	100	4 ± 2	$25\,882 \pm 6\,253$
	Square Taken	60	100	100	4 ± 1	$17\,901 \pm 5\,185$
GPT-OSS-20B	**Aggregate**	**76**	**100**	**100**	$\mathbf{3 \pm 2}$	$\mathbf{20\,021 \pm 6\,272}$
Llama-4-Maverick	Alt. Turns	100	100	100	1 ± 1	$10\,097 \pm 1\,108$
	Draw	100	100	100	3 ± 1	$16\,033 \pm 1\,057$
	Game Over	60	100	100	4 ± 1	$29\,227 \pm 2\,973$
	Player Wins	80	100	100	3 ± 1	$25\,705 \pm 4\,178$
	Square Taken	0	0	0	5 ± 0	$12\,887 \pm 586$
Llama-4-Maverick	**Aggregate**	**68**	**80**	**80**	$\mathbf{3 \pm 2}$	$\mathbf{19\,521 \pm 7\,616}$
Llama-4-Scout	Alt. Turns	20	60	100	5 ± 1	$12\,855 \pm 1\,511$
	Draw	20	60	100	4 ± 2	$16\,442 \pm 2\,227$
	Game Over	0	0	0	5 ± 0	$24\,769 \pm 828$
	Player Wins	0	0	0	5 ± 0	$24\,147 \pm 1\,432$
	Square Taken	0	0	0	5 ± 0	$12\,930 \pm 669$
Llama-4-Scout	**Aggregate**	**8**	**24**	**40**	$\mathbf{5 \pm 1}$	$\mathbf{18\,156 \pm 5\,581}$
qwen3-coder-30b-a3b	Alt. Turns	100	100	100	3 ± 1	$10\,975 \pm 417$
	Draw	100	100	100	3 ± 1	$14\,620 \pm 1\,111$
	Game Over	0	0	0	5 ± 0	$28\,994 \pm 2\,069$
	Player Wins	100	100	100	3 ± 1	$22\,607 \pm 1\,053$
	Square Taken	0	0	0	5 ± 0	$11\,930 \pm 1\,032$
qwen3-coder-30b-a3b	**Aggregate**	**60**	**60**	**60**	$\mathbf{4 \pm 1}$	$\mathbf{18\,145 \pm 7\,186}$
qwen3-coder-480b-a35b-instruct	Alt. Turns	100	100	100	2 ± 1	$9\,262 \pm 802$
	Draw	100	100	100	2 ± 1	$14\,036 \pm 2\,046$
	Game Over	40	90	100	4 ± 1	$27\,187 \pm 1\,539$
	Player Wins	100	100	100	1 ± 1	$18\,912 \pm 2\,662$
	Square Taken	40	90	100	4 ± 1	$11\,248 \pm 637$
qwen3-coder-480b-a35b-instruct	**Aggregate**	**76**	**96**	**100**	$\mathbf{3 \pm 2}$	$\mathbf{17\,378 \pm 6\,723}$

5.0.2. RQ2. Practitioner assessment via structured interviews

We conducted the semi-structured interview with part of the development team at Osoco, composed of two senior software engineers, with extensive experience in software development and testing. In Table 2 we summarize the main findings from the interview by reporting the coded categories from the practitioners' responses (i.e., P1 and P2), along with their convergence status. The Likert scale is from 1 (strongly disagree) to 5 (strongly agree), while $*$ indicates the cases in which the participant did not provide a specific Likert score, and that score was encoded based on a rubric [20]. The convergence status indicates whether the two practitioners agreed on the assessment of each category, with *Strong* indicating agreement, *Weak* indicating partial agreement, and *Divergent* indicating disagreement. In

Table 2
The coding categories extracted from the semi-structured interview for the two practitioners (P1 and P2) and the convergence of their opinions. $*$ indicates a Likert score derived from the qualitative assessment of the participant.

Code	Definition	P1	P2	Convergence
IndustrialValue (Likert)	Confidence to use the approach (the agentic pipeline) in an industrial setting	5	5	Strong
TDDLoopTrust	Trust in the pipeline's test-driven verification loop	✓	✓	Strong
LocalOpenWeightModels	Free local open-weight models are better for adoption	✓	-	P1 only
Correctness (Likert)	The code adheres to the specification	3	5	Divergent
BPIdiomaticness	Code is functionally correct but not idiomatic BP	✓	✓	Strong
ResponsibilityCoupling	The b-thread is implementing more than one behavior	✓	-	P1 only
CodeComprehensibility (Likert)	The implementation of the behavioral logic is easy to comprehend	3*	3	Strong
Maintainability (Likert)	Confidence to modify the code if a rule changes	2*	3*	Weak
RegenerateViaTDD	Confidence in the testing mechanism for regenerate a new code using the TDD pipeline	✓	✓	Strong
TestGeneration	For new behaviors tests must be created from the spec first	-	✓	P2 only
Reliability (open-ended)	The pipeline is perceived to be reliable to generate code	✓	✓	Strong
GeneralizationScope	There can be limitations when applied to more complex behaviors	✓	✓	Strong
RegressionTestingGap	Unit-testing one b-thread in isolation cannot detect regression testing failures	✓	✓	Strong
LanguageChoice	The programming language affects the results	-	✓	P2 only
BPModularityTracing	BP's modular small-pieces structure eases tracing/debugging in large systems	-	✓	P2 only
SpecDesign (open-ended)	The specification format is adequate to the generation task	✓	✓	Strong

the remaining cases, only one of the two practitioners provided a response for that category.

Specifically, we defined six main categories: *IndustrialValue*, *Correctness*, *CodeComprehensibility*, *Maintainability*, *Reliability*, and *SpecDesign*. Preliminary feedback from the two interviewed practitioners suggests the approach has strong potential industrial value. They also perceived the approach to be reliable, given the trust of the fix-test-loop, but the facility of adoption can be strengthened by the usage of local open-weight models. Also, the generated code is not idiomatic BP, and the maintainability of the generated code is limited, as it is not easy to modify the code if a rule changes, but better to regenerate it from scratch using the TDD pipeline. The practitioners also reported that the approach is reliable, but can have limitations when applied to complex behaviors, along with the lack of regression testing verification with the current settings.

Despite this being a positive outcome, there are some limitations that need to be addressed before the approach can be adopted in practice.

6. Discussion

The results of our industrial experience report show that the proposed Test-Driven pipeline for generating behavioral models is promising and well-received by the interviewed practitioners. This grounds the claim directly to your specific sample. However, there are some limitations that need to be addressed before the approach can be adopted in practice as a full-fledged solution.

Reliability of the generated code. The models are able to produce code that is functionally correct, but they are producing non-idiomatic SBP code. This is also dependent on the model; When asking a cross-evaluation in the cool-off phase to P1, to the code implementation evaluated by P2, he rated the code generated by *GPT-OSS-120B* as more idiomatic than the code generated by *gemma-4-26b-a4b-it*. The first created a *b-thread* for handling part of the input parsing, while the second provided all the parsing as procedural code. Also, when manually checking some of the generated code (*Square Taken* from *GPT-OSS-120B*), we found that in one iteration the model generated a b-thread without relying on the *BPpy* library, but instead using a custom implementation. This means that a more strict validation of the generated code is needed to ensure the correctness and idiomaticity of the generated code. A solution can be adding a static analysis phase to the pipeline, to ensure the usage of the correct libraries or using an LLM-as-judge approach to validate the adherence to the SBP paradigm.

Language dependence. The reliability of the generated code is heavily dependent on the programming language and the libraries used. Despite opting for a widely used language (Python), in the initial test of the approach we had several problems with *BPpy*, since the models seem to be not very

familiar with the library directives. This is also reflected in the interview, where P2 reported that the programming language affects the results. An open-question in that direction is to experiment with better-known libraries, such *BPjs*, or, as suggested by P2, evaluating the usage of a DSL language for BP that can be translated in actual code.

Regression testing. We restricted our evaluation to the generation of single b-threads, allowing us to use unit tests to verify the correctness and guide the models during the generation. However, the practitioners reported that this is not enough to ensure the reliability of the generated code, since in a real-world scenario, the generated b-threads will be integrated with other b-threads, and in these cases the regression tests are fundamental to ensure that the new b-thread does not break the existing ones. The open challenge is to integrate this in the current approach as a secondary phase: We can have several pipelines, one for single b-thread, merged all together in a final iteration where all the generated b-threads are tested together.

The implications of the proposed approach are the following. For *practitioners*, the proposed approach is promising for generating single b-threads, but it is not yet production-ready. The test-verification-loop is a key factor for the practitioners' trust in the approach, but the maintainability of the generated code and the lack of regression testing are important limitations that need to be addressed. For *researchers*, our findings highlight several open challenges such as (i) benchmarking and evaluating this kind of approaches on less known programming paradigms, such as BP, (ii) how to improve the reliability when using LLMs for less known languages and libraries, and (iii) techniques for ensuring the idiomaticity of a specific programming language.

7. Threats to Validity

Construct Validity. To measure the effectiveness of the approach, we used the pass@k which is a state-of-the-art metric for code generation tasks. However, this metric may not fully capture other aspects such as test coverage and idiomaticity. We mitigate this by complementing the quantitative evaluation with a qualitative assessment through the semi-structured interview. Also, the effectiveness of the approach could be influenced by the specific characteristics of the LLMs used. To mitigate this threat, we used a diverse set of LLMs with different sizes to avoid bias towards a specific model.

Internal Validity. The analysis conducted in the semi-structured interviews may be subject to human bias. To mitigate this threat, we employed a coding scheme to categorize the responses validated and discussed by two of the authors. The participants authored the specifications used in the study, and the interviewer was the one involved in the development of the pipeline. This may have introduced bias in the responses. This threat was mitigated by ensuring that the interview questions were open-ended and focused on the participants' experiences and perceptions, rather than their prior knowledge of the approach. Additionally, the interviewer must have been necessarily familiar with the approach to be able to answer to the participants' questions about the approach.

External Validity. The external validity of our study may be limited by the specific industrial context in which the evaluation was conducted. Specifically, the number of practitioners involved was limited to two, and the behavior are related to a single domain (Tic-tac-toe). While it is true that the findings may not be generalizable in other contexts, we believe that the study provides still valuable insights. However, we acknowledge that the findings are still preliminary and exploratory.

8. Conclusion

Scenario-based programming allows designing and implementing modular complex behaviors by defining independent interacting scenarios. However, its adoption in industry has some limitations, due to the complexity of designing and implementing the behaviors threads encapsulating the event flows. In this paper, we proposed an industrial case study of a test-driven based approach for b-thread generation, using a generate-and-test loop to guide LLMs and ensure the correctness and reliability of the generated code. We evaluated the approach quantitatively, measuring the functional correctness

of the generated code, and qualitatively, gathering preliminary feedback from two practitioners via a semi-structured interview. These initial results suggest that the approach is promising, but there are still some limitations that need to be addressed before it can be adopted in practice. The future work will focus on addressing the limitations identified in the study, and on extending the approach to support the generation of multiple interacting b-threads, to allow the generation of complex behaviors and full applications.

Data Availability

The replication package with the scripts, interview guide and data used in this study is publicly available [20].

Acknowledgment

This work has been partially supported by the Spanish Ministry of Science and Innovation under Grant No. 2024/00416/002. We thank the Osoco team for their valuable support and collaboration in the study.

References

[1] K. Beck, Test-driven development: by example, Addison-Wesley Professional, 2003.
[2] B. Papis, K. Grochowski, K. Subzda, K. Sijko, Experimental evaluation of test-driven development with interns working on a real industrial project, IEEE Transactions on Software Engineering 48 (2020) 1644–1664.
[3] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, S. K. Lahiri, LLM-based test-driven interactive code generation: User study and empirical evaluation, IEEE Transactions on Software Engineering (2024).
[4] N. S. Mathews, M. Nagappan, Test-driven development and LLM-based code generation, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1583–1594.
[5] S. Piya, A. Sullivan, LLM4TDD: best practices for test driven development using large language models, in: Proceedings of the 1st International Workshop on Large Language Models for Code, 2024, pp. 14–21.
[6] S. Piya, A. Samadi, A. Sullivan, Is more or less automation better? an investigation into the LLM4TDD process, in: 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), IEEE, 2025, pp. 161–168.
[7] D. Harel, R. Marelly, Come, let's play: scenario-based programming using LSCs and the play-engine, volume 1, Springer, 2003.
[8] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., Program synthesis with large language models, arXiv preprint arXiv:2108.07732 (2021).
[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).
[10] D. Harel, G. Katz, A. Marron, S. Szekely, On augmenting scenario-based modeling with generative ai, arXiv preprint arXiv:2401.02245 (2024).
[11] T. Yaacov, A. Elyasaf, G. Weiss, Boosting llm-based software generation by aligning code with requirements, in: 2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW), IEEE, 2024, pp. 301–305.
[12] T. Yaacov, Bppy: Behavioral programming in python, SoftwareX 24 (2023) 101556.
[13] A. Berzack, G. Katz, On integrating large language models and scenario-based modeling for improving software reliability, in: Proceedings of the 14th International Conference on Model-

Based Software and Systems Engineering - Volume 1: MODELSWARD, INSTICC, SciTePress, 2026, pp. 21–32. doi:10.5220/0014211100004058.

[14] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia, R. Nallapati, M. K. Ramanathan, D. Roth, B. Xiang, ReCode: Robustness evaluation of code generation models, in: A. Rogers, J. Boyd-Graber, N. Okazaki (Eds.), Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Toronto, Canada, 2023, pp. 13818–13843. URL: https://aclanthology.org/2023.acl-long.773/. doi:10.18653/v1/2023.acl-long.773.

[15] F. Rabbi, Z. Ding, J. Yang, A multi-language perspective on the robustness of llm code generation, Empirical Software Engineering 31 (2026) 146.

[16] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, et al., Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, in: International Conference on Learning Representations, volume 2025, 2025, pp. 66602–66656.

[17] OpenAI, gpt-oss-120b & gpt-oss-20b model card, 2025. URL: https://arxiv.org/abs/2508.10925. arXiv:2508.10925.

[18] Meta AI, Llama 4: Official documentation and model cards, https://ai.meta.com/blog/llama-4-multimodal-intelligence/, 2026. Accessed: 2026-06-17.

[19] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al., Qwen3 technical report, arXiv preprint arXiv:2505.09388 (2025).

[20] G. Rosa, D. Moreno-Lumbreras, G. Robles, J. M. Gonzalez-Barahona, Replication Package for "From Specifications to Behavioral Threads: An Industrial Evaluation of Test-Driven Code Generation", 2026. Https://tinyurl.com/mr3cw78n.

[21] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical software engineering 14 (2009) 131–164.