On Integrating Large Language Models and Scenario-Based Modeling for Improving Software Reliability

Ayelet Berzack and Guy Katz
The Hebrew University of Jerusalem, Jerusalem, Israel
{*ayelet.berzack, g.katz*}*@mail.huji.ac.il*

Keywords: Scenario-Based Modeling, Rule-Based Specifications, Large Language Models, Formal Verification.

Abstract: Large Language Models (LLMs) are fast becoming indispensable tools for software developers, assisting or even partnering with them in crafting complex programs. The advantages are evident — LLMs can significantly reduce development time, generate well-organized and comprehensible code, and occasionally suggest innovative ideas that developers might not conceive on their own. However, despite their strengths, LLMs will often introduce significant errors and present incorrect code with persuasive confidence, potentially misleading developers into accepting flawed solutions.

In order to bring LLMs into the software development cycle in a more reliable manner, we propose a methodology for combining them with "traditional" software engineering techniques in a structured way, with the goal of streamlining the development process, reducing errors, and enabling users to verify crucial program properties with increased confidence. Specifically, we focus on the Scenario-Based Modeling (SBM) paradigm — an event-driven, scenario-based approach for software engineering — to allow human developers to pour their expert knowledge into the LLM, as well as to inspect and verify its outputs.

To evaluate our methodology, we conducted a significant case study, and used it to design and implement the Connect4 game. By combining LLMs and SBM we were able to create a highly-capable agent, which could defeat various strong existing agents. Further, in some cases, we were able to formally verify the correctness of our agent. Finally, our experience reveals interesting insights regarding the ease-of-use of our proposed approach. The full code of our case-study will be made publicly available with the final version of this paper.

1 INTRODUCTION

Since their appearance, large language models (LLMs) have dramatically changed the way complex software is designed and maintained. Modern LLMs, such as GPT-5, are able to quickly produce high-quality code, and have become integral tools in the software engineering toolkit (Du et al., 2024). These models contribute to many aspects of the software life-cycle, including prototyping, debugging, deployment and interpretability; and their use is likely to increase and expand in coming years (Haque, 2025).

Despite these impressive advantages, the rise in use of LLMs also presents unique challenges. Most notably, LLMs often err — for example, they might produce code that operates incorrectly in corner cases — and they often do so with persuasive confidence. The increasing reliance of human engineers on these tools could thus lead to the deployment of faulty code, making the whole process potentially unreliable for complex, safety-critical applications. Recent studies have already shown that LLMs sometimes perform poorly on certain tasks that involve interdependent logic, such as class-level code generation, where multiple methods must interact correctly (Du et al., 2024). Thus, novel methodologies for leveraging LLMs in a way that results in reliable software are sorely needed.

Our work here proposes one such new engineering methodology, aimed at mitigating the risks involved by using LLMs as part of the development cycle, while still retaining the benefits they afford. Our proposed methodology is hybrid, in the sense that it combines human guidance with automated, LLM-based code generation. The idea is to allow human domain experts to pour their knowledge into the LLM in order to guide it, but at the same time ensure that the LLMs outputs are interpretable, and verifiable, by the domain expert. This is achieved using structured prompts, iterative refinement, and human-in-the-loop feedback to reduce hallucinations and improve logical consistency.

One major challenge in this kind of approach is

Berzack, A. and Katz, G.
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 (MODELSWARD 2026)*, pages 21-32
ISBN: 978-989-758-798-6; ISSN: 2184-4348

that it may be difficult for the human expert to assess the correctness of code generated by the LLM. To mitigate this difficulty, we advocate the use of Scenario-Based Modeling (SBM) (Harel et al., 2012b), where complex software is expressed as a collection of inter-dependent scenarios of desirable or undesirable behavior. The advantages of using SBM in this context are two-fold. First, it allows for a well-structured way for the domain expert to incrementally refine the specifications provided to the LLM, until the desired outcome is reached. Second, scenario-based models are known to be more readily interpretable, both by humans and by automated analysis techniques, such as model checking (Marron et al., 2018); and this allows to more easily, and semi-automatically, inspect the LLM's outputs and gain confidence in their correctness.

Building on this foundation, we introduce a structured, SBM-based methodology for guiding LLMs in the development of complex software. The process begins with the developer defining high-level behavioral goals and decomposing them into discrete, modular scenarios, each representing a specific requirement or constraint. Prior to implementation, the LLM is provided with a preamble consisting of relevant background information — both about the system being developed and about Scenario-Based Modeling itself, which is not widely represented in existing online resources. Implementation then proceeds incrementally: for each scenario, the developer engages in an iterative process with the LLM to generate a corresponding scenario object. This involves crafting focused, well-scoped queries that include relevant assumptions, context, and known events. The LLMs output is reviewed for correctness before integration — typically through manual inspection, but sometimes supported by explicit or symbolic formal verification to ensure key properties are satisfied. The LLM plays an active role throughout this process: it generates code, proposes refinements, and can even assist in debugging. When issues are discovered during testing or verification, the developer presents them to the LLM, which can help diagnose the problem and suggest or implement corrections. In this workflow, the LLM acts as a proactive coding assistant, while the developer guides the process, supplies domain knowledge, and ensures correctness.

In order to evaluate our approach, we conducted an extensive case-study. Specifically, we created an agent for playing the popular board game, *Connect4*, and were able to achieve excellent results. We consider this to be strong evidence of the potential effectiveness of our hybrid methodology.

The rest of this paper is organized as follows. In Section 2 we briefly introduce the Connect4 game, which we used as a case-study. Next, in Section 3 we provide necessary background on Large Language Models and Scenario-Based Modeling. In Section 4 we go into detail about our proposed methodology. In Section 5 we describe our case study and show how the methodology was applied to build a Connect4 agent. Section 6 presents a detailed evaluation of the agent's performance, formal correctness, and development process. In Section 7 we discuss the limitations of our approach, followed by a discussion of related work in the domains of LLM-based development and scenario-based modeling in Section 8. Finally, we conclude and outline directions for future work in Section 9.

2 INTRODUCING Connect4

Connect4 is a popular board game, where two players take turns dropping colored discs into a vertical 7×6 board, aiming to be the first to form a line of four discs of their own color (typically red and yellow), either horizontally, vertically, or diagonally (Figure 1). Connect4 is a partially-solved game: it has been formally proven that the first player can always win with perfect play (Allis, 1988). However, existing perfect solutions rely on brute-force search, and it is presently unknown how to solve the 7×6 variant of the game using rules and heuristics comprehensible to humans. Thus, creating an agent whose behavior a human can readily understand remains an interesting problem, which we seek to tackle here.

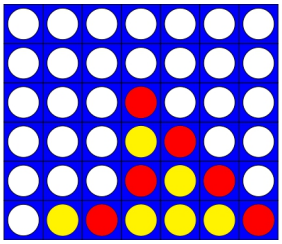

Figure 1: A Connect4 board game sample where red wins with a diagonal sequence of 4 discs.

In order to evaluate our work, we used our proposed methodology to create an agent for playing Connect4. Our goal was to investigate whether strategic game-play could be effectively constructed by a large language model when guided by a structured SBM-based development process. We sought to create, with the help of an LLM, a scenario-based agent that plays "yellow" in Connect4, and which achieves results comparable with state-of-the-art solutions. As part of our methodology, we first had the LLM gener-

ate scenarios that described the game rules; and later, also scenarios that implemented strategies. We repeatedly evaluated the agent's capabilities using both testing and formal verification, and iteratively query the LLM to modify and correct certain aspects of the scenarios. The modular nature of SBM facilitates this process, allowing the human-in-the-loop to focus on one particular aspect of the game each time. The LLM proved highly beneficial in translating natural language descriptions into working code — specifically, turning scenario descriptions into scenario objects. This not only saved development time, but also led to the generation of novel code structures that we might not have considered independently. Our ability to explicitly verify the resulting model further strengthened the process: it enabled us to confirm that no rules were violated and provided formal guarantees that the agent would win under certain constrained move sequences.

Our agent achieved remarkable results, demonstrating the effectiveness of our hybrid methodology. It outperformed three distinct AI-based Connect4 opponents, all employing *minimax* and *alpha-beta pruning*, one of which boldly claims the title "Unbeatable AI". The first is the "Pro Player" available at Tleemann's Connect4 site (Tleemann, 2025), the second is the "Unbeatable AI" hosted at Websim (Doe, 2025), and the third is an AI player developed by Keith Galli, available on his GitHub repository (Galli, 2025). In addition, although the agent is not always guaranteed to win, we used formal verification to prove that it will always win under several fixed sequences of the first x moves, providing valuable partial correctness guarantees. These results highlight the promise of combining LLM-driven code generation with scenario-based modeling to develop robust, maintainable, and high-performing systems.

3 BACKGROUND AND DEFINITIONS

3.1 Large Language Models (LLMs)

Large Language Models (LLMs) are deep learning models trained on vast textual corpora to predict and generate coherent natural language. Recent LLMs, such as GPT-5 (Wang et al., 2025), PaLM (Chowdhery et al., 2022), and Claude (Priyanshu et al., 2024), exhibit impressive generalization capabilities. They can perform a wide range of tasks — including natural language generation, translation, code synthesis, and debugging — often with little or no task-specific training. This flexibility has made them powerful tools in software engineering workflows.

When applied to software development, LLMs can suggest code completions, generate functions from descriptions, refactor logic, and explain existing code. Their growing integration into IDEs and developer workflows has made them valuable assistants in everyday programming (Cursor, 2023).

However, LLMs are inherently probabilistic and lack grounded semantic understanding. They may confidently generate incorrect or unexecutable code — commonly referred to as *hallucinations* — and their outputs are often difficult to verify or formally constrain. These limitations raise concerns when using LLMs in safety-critical, logic-intensive, or formally verifiable domains, motivating hybrid approaches like the one explored in this paper.

3.2 Scenario-Based Modeling

Scenario-Based Modeling (SBM) is a modeling paradigm in which the behavior of a system is specified in terms of interacting, independent scenarios, each representing a partial view of the system's functionality — such as a requirement, a use case, or a constraint (Harel et al., 2012b). These scenarios are implemented as *scenario objects* (also referred to as *scenario threads*), which are executed concurrently and coordinate by declaring three types of event-related intentions: requesting events to occur, waiting for events to occur, and blocking events from occurring.

Execution in SBM progresses in discrete synchronization points. At each point, all active scenario objects announce their current declarations. The SBM runtime environment then selects one event to trigger — among those that are requested by at least one scenario object, and not blocked by any. All scenario objects that requested or waited for the selected event then resume execution, while the others remain suspended; and the process then repeats at the next synchronization point.

Events. An *event* in SBM is an atomic action or occurrence in the system. Events are typically represented by simple data objects (e.g., a name or dictionary). Only events that are requested and not blocked can be selected for triggering by the engine.

Scenario-Objects. A *scenario object* defines a self-contained behavior. Each scenario object yields control at synchronization points, specifying its interest in events. These objects are designed to be loosely coupled: they interact solely through their event dec-

larations and are unaware of each other's internal state.

Synchronization Points. These are moments in the model's execution where all scenario objects pause and submit their event declarations. The runtime engine then selects a single event that satisfies the collective constraints. Once an event is selected, the relevant scenario objects proceed, maintaining event-driven coordination.

Behavioral Composition. The overall system behavior emerges as a composition of the independent contributions of all scenario objects. Because scenario objects are modular and communicate via events, developers can incrementally add or remove behaviors, often without needing to modify existing code (Harel et al., 2012b). This facilitates both maintainability and scalability.

In this paper, we use *BPpy* — a Python-based implementation of Scenario-Based Modeling (Yaacov, 2023) — to illustrate the concepts and run simulations. Below is a simple BPpy example that demonstrates the basic mechanics of SBM: two scenario objects requesting "Hot" and "Cold" water, respectively, and a third scenario object ensuring temperature stability, by forcing that hot and cold water be added alternately. The runtime selects and prints one event at a time:

```
from bppy import *

@b_thread
def add_hot_water():
 while True:
 yield bp.sync(
 request=BEvent("Hot"))

@b_thread
def add_cold_water():
 while True:
 yield bp.sync(
 request=BEvent("Cold"))

@b_thread
def stabilize():
 while True:
 yield bp.sync(
 waitFor=BEvent("Hot"),
 block=BEvent("Cold"))
 yield bp.sync(
 waitFor=BEvent("Cold"),
 block=BEvent("Hot"))

bprog = SBMrogram(
 bthreads=[add_hot_water(),
     add_cold_water(),
 stabilize()]
)
bprog.run()
```

When this model is executed, the two requested events are interleaved in an infinite loop. Because SBM relies on synchronization rather than imperative control flow, system behavior is the emergent result of collaborative event selection and thread progression. A depiction of the resulting model, this time as a transition system, appears in Fig. 2

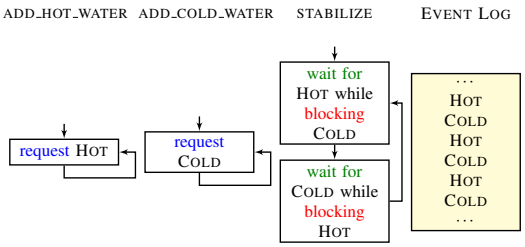

Figure 2: A scenario-based model of a system that interleaves "Hot" and "Cold" events.

BPPy also ships with a built-in model checker (Yaacov, 2023), which supports explicit and symbolic modes. In the explicit mode, assertions provided by the user are verified using a DFS traversal of the state space. In symbolic mode, the model checker attempts to curtail the search space to expedite verification; and uses the NuSMV model checking tool as a backend (Cimatti et al., 2002). We used this built-in model checker in our experiments, as described later.

4 METHODOLOGY

In order to allow engineers to harness the capabilities of LLMs in a reliable manner, we propose the following methodology for designing complex software, using scenario-based modeling. The methodology emphasizes incremental development, careful planning, and active collaboration between the developer and the LLM. The key insight is that while LLMs are effective in generating localized behaviors, they struggle with large-scale planning or modifying code holistically. Therefore, we recommend a human-guided, modular workflow, outlined below.

The workflow begins with an initial planning phase, where the developer defines the desired behaviors and ensures the LLM has the necessary background knowledge. Once this foundation is in place, the remaining scenario objects are implemented iteratively, one at a time, using a query-review-refine cycle until the system is complete.

- **Step 1: Define the Scenarios.** Begin by thoroughly considering the overall goals and intended behavior of the model. Go over the system speci-

fication, and for each requirement or rule, record it as an individual scenario. This process should feel intuitive, with no need to think about implementation details or code at this stage. Each scenario will later be implemented as a separate scenario object in the model. Each scenario should have a narrow, well-defined responsibility. The more granular and modular the components, the easier they are to implement and verify. Once a preliminary set is formed, present it (in free text) to the LLM, and ask whether it identifies any missing responsibilities. This process helps the LLM understand the scope of the system being developed and also helps the developer organize their thoughts. By the end of this step, there should be a clear, high-level structure for the SB model.

- **Step 2: Provide Background Knowledge to the LLM.** Before beginning the implementation, it is essential to equip the LLM with any relevant background knowledge that it may lack. This includes a general overview of Scenario-Based Modeling, which is currently underrepresented in online resources — resulting in limited familiarity on the part of the LLM compared to more mainstream programming paradigms. It is also important to provide domain-specific background related to the application being developed, along with examples that can help guide the generation process.
- **Step 3: Incremental Scenario-Thread Development and Refinement.** Begin implementation by asking the LLM to generate one scenario object at a time. For each scenario, provide a clear description of the required behavior and responsibility, including any assumptions or known events. Avoid asking the LLM to produce multiple components at once — this often results in bloated, inconsistent, or confused output. After generating each scenario object, manually inspect the logic and correctness of the code before integrating it into the larger SB model. Because scenario objects are loosely coupled, this stepwise integration allows the system to be incrementally tested and debugged. If a scenario object does not meet the specification or requires adjustment, provide targeted feedback to the LLM and iterate on its response until the result is satisfactory.

Revising Previously Implemented Scenario Objects. Previously implemented scenario objects can be modified when necessary, either to refine behavior or to adapt to new requirements. This process mirrors the approach used in Step 3: the engineer queries the LLM with the existing scenario object and a description of the required changes. When more extensive changes are needed — such as modifying multiple scenario objects — we found it significantly more effective to handle each change individually. Rather than asking the LLM to revise several scenario-threads at once, we achieved better results by querying it with a single code block and a focused instruction at a time. This approach aligns better with the model's limitations and reduces the risk of introducing new errors.

Testing and Verifying. Throughout the process, treat the LLM as a coding partner rather than a fully autonomous agent. Frequently ask clarifying questions, present it with counterexamples, and challenge its assumptions. In return, the LLM will prove especially helpful in generating ideas, validating small design decisions, and drafting clean Python code. At any stage, if there is a property that should be verified, apply off-the-shelf verification tools for that individual object.

In summary, a successful collaboration with LLMs for SB modeling requires the engineer to take an active planning role, decompose behaviors clearly, and work with the LLM incrementally. This results in modular, verifiable models and avoids the common pitfalls of over-relying on the LLM to understand or construct complex architectures independently.

5 CASE STUDY: Connect4 BEHAVIORAL MODEL

5.1 Applying the Methodology: Implementing Core Game Mechanics

In our case study we sought to address the following research questions:

- **RQ1:** Can LLMs, when guided using structured prompts and SBM, generate correct and modular software models for complex systems?
- **RQ2:** Can an LLM-guided SBM agent achieve performance comparable to state-of-the-art agents in a non-trivial domain?
- **RQ3:** How convenient is it for an engineer to combine SBM and LLMs?

To that end, we applied our methodology to a complex case study: the task of generating a Connect4 game-playing agent. Our first goal was to create an agent that "understands" the game rules — i.e., it

plays valid moves, recognizes when the game ends, and adheres to the rules of the game. The second goal was to enhance this agent by implementing various strategies to improve its gameplay. In the remainder of this section, we demonstrate how our methodology helped us achieve the first goal, and in Section 5.2, we focus on the second goal. We initially used OpenAI's GPT-4 (with a low-temperature setting) and later transitioned to Claude (Anthropic, 2023) for longer sessions, taking advantage of its capabilities for more extensive development. Additionally, we incorporated the Cursor IDE (Cursor, 2023) for further flexibility and seamless integration during the longer sessions.

Step 1: Define the Scenarios. The SB model we seek to develop should represent a core Connect4 agent, which does not yet posses any game-playing strategies. Thus, the agent can select an arbitrary move in each turn, as long as it is allowed. We formulated the following scenarios:

- **Board Representation.** The game board is a 7 by 6 grid, totaling 42 slots. The board is considered to be vertical (i.e., one side is the "top"), and game discs from the top of a selected column and fall to the lowest available slot therein. Once a column is full, no more discs can be placed there.
- **Player Roles.** There are two players — red and yellow — each placing colored discs on their turn. Our agent plays yellow, and its opponent plays red.
- **Turn Alternation.** Players alternate turns, starting with yellow.
- **Winning.** A player wins by placing four of their discs in a horizontal, vertical, or diagonal line.
- **Draw.** If the board is full with no winner, the game ends in a draw.

Step 2: Provide Background Knowledge to the LLM. To initiate the implementation, we provided the chatbot with a preamble describing the Scenario-Based Modeling paradigm, along with concrete examples. We instructed it to assist in building the Connect4 model using BPpy, while explicitly stating that it should not make assumptions about rules not provided. Since the game of Connect4 is well known, we did not need to provide detailed background; instead, we verified the LLMs familiarity and ensured that its understanding of the game's rules aligned with ours.

Step 3: Incremental Scenario-Thread Development and Refinement. In this key step, we began iteratively querying the LLM to produce the individual scenarios that represent the game rules. We instructed the LLM to implement a scenario object corresponding to each described behavior. We then reviewed the LLMs output. If the result was flawed, we iteratively refined the query and provided corrective feedback until it met our standards.

To illustrate this process, consider the implementation of the player-roles scenario for the red player. The initial query was:

> *The second player is the user. On their turn, they input the column where they want to place a red disc. Just like the agent, they always try to place a disc in that column.*

The LLM initially generated a scenario object `user_player`, but it misunderstood the intended behavior. Initially, the LLM attempted to determine the lowest available slot in the selected column and requested that specific placement. We corrected this by clarifying that this behavior was handled by other scenario objects. The `user_player` object's role was simply to request all possible placements in the selected column. We also reminded the LLM to use a global list of red placement events. The corrected version is shown below:

```
all_red_placements =
 [place_red(row, col)
 for col in range(num_cols)
 for row in range(num_rows)]

@bp.thread
def user_player():
 while True:
 col = int(input(
 "Enter column number for placing red
     dics"))

 requests =
 [place_red(row, col)
 for row in range(num_rows)]

 yield bp.sync(request=requests)
```

We repeated this process for each of the aforementioned game rules; and the LLM successfully translated each of them into one or more scenario objects. For instance, the rule governing valid disc placement was assigned to a scenario object called `valid_placement`, which is responsible for blocking illegal moves, such as placing a disc in a full column or at invalid row indices.

This example illustrates how the methodology fosters productive collaboration between the developer and the LLM: the developer defines precise

responsibilities and verifies correctness, while the LLM generates the implementation within those constraints.

5.2 Extending the System: Strategy Implementation and Verification

Once the base game was functional, we moved on to the more challenging task of designing and implementing game strategies. We began by asking the LLM to list generally useful strategies, based on its background knowledge. We then prompted it to implement these strategies, one at a time. We started with relatively simple approaches, such as prioritizing the middle column and blocking immediate threats (opponent wins in the next move); and later progressed to more complex strategies, like those involving fork intersections. As an additional source of background knowledge, we manually extracted useful strategies from the literature (Allis, 2025; Allis, 1988; Galli, 2025), and instructed the LLM to implement them. As the strategies grew more complex, the LLM required more fine-grained guidance, but was still able to implement them successfully through well-scoped queries.

After implementing each batch of strategies, we periodically ran the BPPy verifier to identify scenarios where the red player could still win. In one such instance, the yellow player missed an obvious win. We used this insight to query the LLM to implement win-detection scenario objects, which would prioritize placing a winning disc when three aligned yellow discs were already present. Throughout this iterative process, we repeatedly:

1. Ran the verifier to find counterexamples.
2. Presented the red-win trace to the chatbot, and asked it how the current strategy could be improved.
3. Confirmed the LLM's diagnosis (or manually diagnosed the problem if the LLM failed), and then queried the LLM to implement a solution.

For example (Figure 3), in one case the verifier generated a trace with a diagonal fork: three red discs forming a diagonal with empty slots at both ends. Yellow could block only one, allowing red to win on the next turn. While the LLM could not identify the issue independently, we explained to it this new type of fork discovered and it readily implemented a fork-prevention strategy.

One useful observation is that the structure of SBM makes it straightforward to add test-cases as scenarios that force a particular trace (by blocking other events); and these scenarios can later be used to check whether the (undesirable) trace is no longer feasible. This allowed us to create a regression test-suite as we progressed.

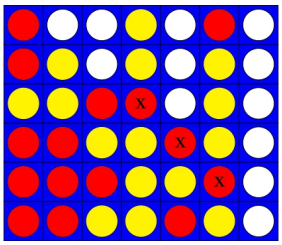

Figure 3: From the trace of a counterexample found by the verifier containing a red diagonal fork.

5.3 Extending SBM with New Features

Throughout our work on the case study, we encountered two challenges that proved difficult to overcome using the existing SBM idioms and semantics.

First, we observed that in some corner cases, certain strategies would block a potential move by the agent — for example, because that move could lead to a later fork, and a consequent loss. However, the same move would actually complete a sequence of four discs, allowing the agent to win the game. Clearly, in such cases, we would prefer that the request to place the disc to be allowed, as the benefits outweigh the risks. However, according to SBM semantics, each scenario object is self-contained, meaning the object blocking the move is unaware that it should in fact be permitted. Additionally, under SBM semantics, blocking events takes precedence over requesting them. Once an event is blocked, there is no way for another scenario object to trigger it.

Secondly, during the development of the intersection fork scenario objects, we encountered situations where it was required to request events with different *priorities* within a single synchronization point. For example, one case is a situation in which red is placing discs in a way that would simultaneously create two 3-long sequences of discs that could each be extended to winning, 4-long sequences. This is particularly risky when these two sequences intersect, and red places the intersection disc last — so that yellow is able to block only one sequence in the next move, allowing red to win. When such a threat is detected, it can be prevented in advance by placing yellow discs that disrupt the two sequences of red discs at any of their locations; but the best solution is to capture the location of the linchpin, intersection disc before red places it. Thus, while several disc placements should be requested, the intersection disc should be requested with a higher priority. Unfortunately, the standard

SBM semantics have no notion of priorities; and the only existing extension with such a concept assigns priorities to entire scenario objects, and not to individual events (Harel et al., 2011).

We resolved these issues by extending the SBM semantics to provide greater control over event handling. Specifically, we allowed each scenario object to assign an *event-specific priority* to each event that it requests or blocks. The motivation to this idea (which was in fact proposed by the LLM when we presented the difficulty to it) was to allow blocked events to be triggered, provided that the request has a higher priority than the block. The event selection mechanism is then adjusted as follows. An event is selected for triggering if:

- It is requested by at least one scenario object.
- If it is blocked, it is requested with a higher priority than the priority with which it is blocked.
- It is requested with the highest priority among all events that meet the above criteria.

In order to be compatible with SBM's original blocking mechanism and the advantages it affords, we allowed blocked events to be blocked with a priority of ∞; alternatively, this could be thought of as simultaneously allowing "hard blocking" of events that can never be triggered, and "soft blocking" of events that may be triggered if they are requested with a higher priority than that with which they are blocked. Clearly, an event that is hard-blocked can never be triggered. For further details on the precise semantics of these extensions to SBM, which may have broader applications, we refer the reader to the full version of this paper (Berzack and Katz, 2024).

6 EVALUATION

To evaluate our hybrid Connect4 agent, we measured its robustness and capabilities using formal analysis (addressing **RQ1**), and also pitted it against external AI-based Connect4 agents used as benchmarks (addressing **RQ2**). Finally, we documented and assessed our experience interacting with the LLM throughout development (addressing **RQ3**). Naturally, the first two aspects are easier to quantify, whereas the third is more subjective.

6.1 Agent Performance

We evaluated our agent against three strong, AI-based opponents available online: the "Pro Player" (Tleemann, 2025), "Unbeatable AI" (Doe, 2025), and the AI player by Keith Galli (Galli, 2025). In each case, our agent played as the first player (yellow). We note that our agent is deterministic, and so are "Unbeatable AI" and the Keith Galli agent; and so repeating the experiment simply repeated the same game. Conversely, the "Pro Player" has some small degree of non-determinism, and so we had our agent play 10 matches against it. *Our agent was able to win every single match, against all opponents, with no losses or draws.* We regard this as indication that our agent implements a reliable and successful strategy. Figures 4, 5, and 6 show the final board states from matches against the three opponents.

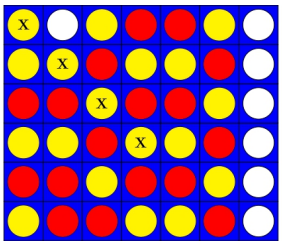

Figure 4: SBM Agent (yellow) vs AI Pro (red).

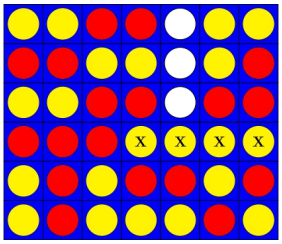

Figure 5: SBM Agent (yellow) vs "Unbeatable AI" (red).

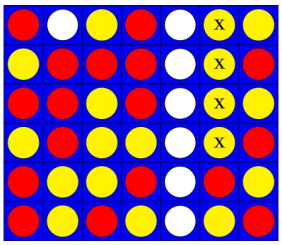

Figure 6: SBM Agent (yellow) vs "Galli's AI" (red).

We thus conclude that the answer to **RQ2** is positive: our results demonstrate that an LLM-guided SBM agent can repeatedly defeat advanced opponents.

6.2 Formal Guarantees

To complement gameplay testing, we used BPPy's explicit model checker (`DFSBMrogramVerifier`) to verify partial correctness. The current state-of-the-art in BP model checking is insufficient for cover-

ing the large state space induced by Connect4; and so, in order to be able to at least partially verify our agent, we studied six, arbitrarily selected *fixed* opening sequences (i.e., a set of fixed initial moves by both players). Restricting the game to start from such a sequence shrunk the search space, and allowed the BPPy model checker to scale to the problem; though it naturally leads to weaker results, covering only certain plays and not the entire space thereof. From each such configuration, we successfully verified that our agent (yellow) was guaranteed to win, regardless of the subsequent moves played by its opponent.

Figure 7 illustrates one such opening sequence, where we observe that yellow has a diagonal odd-threat (the last disc that needs to be placed is in an odd row, and it will inevitably be yellow's turn when it is time to place it); and also that yellow controls most of the middle column. Both of these are strategies employed by our agent, and so such a sequence of moves is quite plausible in a real match.

Figure 8 shows another verified opening configuration, in which yellow forms a classic *fork*, simultaneously creating a diagonal and a horizontal threat. The discs marked with an "X" highlight positions where yellow is poised to win via either line, making it impossible for red to defend against both. This configuration, which occurs in practice due to our agent's strategies, demonstrates the agent's ability to construct complex, multi-threat configurations through modular, verifiable logic.

Finally, Figure 9 presents yet another verified game state, in which yellow is guaranteed to win — as confirmed by the model checker. While the victory is not as immediately apparent from the board state compared to the previous examples, it is clear that yellow's strategic control over the center column plays a critical role in securing the win. This configuration, again reachable by our agent, highlights the advantage of using formal verification to prove that the agent is guaranteed to win from a certain board state, even when the path to victory is not as visually obvious.

These formal results indicate a positive, albeit partial, answer to **RQ1**, showing that the LLM-generated code is verifably correct in many cases. While the agent is not perfect — it may lose, as some edge cases remain unhandled — we demonstrate that a significant subset of the strategy is provably correct under formal analysis.

6.3 Developer Experience

Developing the Connect4 game was both an exciting and challenging experience. Initially, our expectations for the LLM were high — we anticipated it would generate near-perfect scenario-based code. However, we quickly realized this was not the case, partially due to the limited information and examples available on SBM, and especially concerning the BPPy library.

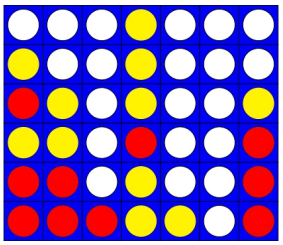

Figure 7: Fixed Moves Example #1.

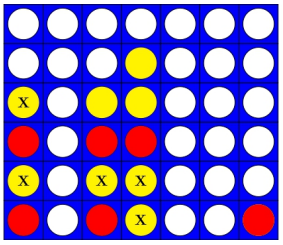

Figure 8: Fixed Moves Example #2.

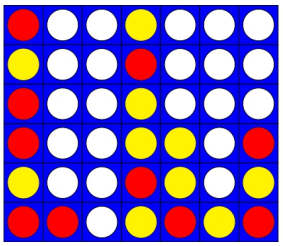

Figure 9: Fixed Moves Example #3.

In the early stages of the case study, we became slightly frustrated with the LLM's tendency to hallucinate functionality. These errors were challenging to overcome, but we soon recognized that providing clear feedback allowed the model to improve rapidly. For instance, the draw-detection logic was generated almost correctly on the first try once we supplied the appropriate context. With iterative refinement and formal verification, we were able to identify and fix errors effectively. One example involved a counterexample (a game trace in which the red player defeats our agent), which revealed a vulnerability related to a diagonal fork. After presenting this issue to the LLM, it successfully generated a mitigation scenario.

A particularly valuable aspect of the LLM was its ability to offer creative solutions when we felt stuck. It was exciting to witness how seamlessly the game strategies could be translated into scenario objects in

our SB model. Progress was evident not only as we played against the agent, but also as we ran simulations, compared the agent's performance with online agents, and validated the system using formal verification. Though there were moments where it felt like we were wasting time with seemingly unproductive conversations with the LLM, we found that when we adhered to the methodology — focusing on one scenario object at a time with precise and clear prompts — the results were highly effective.

In retrospect, we feel confident that future development will be more efficient, thanks to the refined methodology and deeper understanding that we have gained from this experience. Overall, the LLMs were highly effective at transforming behavioral descriptions into code, significantly accelerating the development process. Furthermore, SBM's modularity played a crucial role in isolating logic and supporting verification, which contributed to a more structured and manageable workflow. Thus, with respect to **RQ3** we believe that there is a certain learning curve that must be mastered in order to make the most of our proposed methodology; but that once that phase is reached, it becomes quite convenient and enjoyable. Naturally, much additional experimentation is still needed to fully establish this claim in a broader manner.

7 LIMITATIONS

While our methodology demonstrates a promising approach to developing reliable software with LLM assistance, our work is but a first step, and has several limitations that will need to be addressed in future work. We briefly mention them here.

Scope of Empirical Evaluation. The generalizability of our findings is limited by the Connect4 case-study. While Connect4 is complex, it operates under clear, finite, and well-defined rules, making it highly amenable to exhaustive search and structured Scenario-Based Modeling (SBM) decomposition. The efficacy and reported convenience (**RQ3**) of our methodology may not immediately generalize to more complex, open-ended, or safety-critical domains lacking clear-cut, quantifiable win/loss conditions. Furthermore, the current evaluation's statistical rigor is limited, as the performance testing relied on a few, deterministic opponents and restricted the agent to playing only as the first player.

Constraints on Formal Verification. Our formal correctness guarantees are inherently partial due to the state-space limitations of the current BPPy model checker. The verification results are confined to a small, manually selected set of six fixed opening sequences, which restricts our claims to verification under restricted initial states rather than broad correctness across the full game state space. As BP verification technology continues to improve, this limitation will be alleviated.

8 RELATED WORK

Recent developments in Large Language Models (LLMs) have inspired a surge of interest in using generative models to assist with software engineering tasks. Several works have studied the effectiveness of LLMs in generating program code from natural language prompts (Du et al., 2024; Haque, 2025). However, research increasingly emphasizes that unstructured use of LLMs often results in unreliable or incorrect code. To address this, recent work has proposed frameworks that integrate LLMs into structured development pipelines. For example, Sun et al. (Sun et al., 2024) introduce *Clover*, which adds verification to LLM-generated code using a closed-loop feedback cycle. Similarly, Yixuan et al. (Li et al., 2024) present a framework in which LLMs guide enumerative program synthesis through specification refinement and ranking of candidate implementations. Both emphasize the need for structured workflows when deploying LLMs in safety — or correctness-critical domains.

Our work aligns with recent efforts to structure LLM-based development using scenario-based techniques (Harel et al., 2024). The motivation for focusing SBM is that scenario objects tend to be well-aligned with how humans perceive systems, as well as amenable to formal analysis techniques (Harel et al., 2015).

The connection between LLMs and scenario-based modeling has previously been explored by Yaacov et al. (Yaacov et al., 2024), who argued that structuring code generation around modular scenario threads improves verifiability and traceability. They show how each requirement can be transformed into an independent code module, and how SBM modularity can guide prompt engineering and reduce LLM-induced errors, using both explicit and symbolic verification. Building on this work, we take their research a step further by creating a methodology for combining LLM development with SBM. We demonstrate how this methodology can be applied to develop more complex systems, providing a comparative analysis with existing systems available online. Additionally, we demonstrate how performing formal verifica-

tion on the complex systems under development can help ensure correctness in this setting. Finally, we introduce new features to the SBM semantics (and the BPPy package), facilitating its integration with LLMs and its use in devising complex systems.

On a broader scope, the usefulness and applicability of scenario-based modeling has been demonstrated in various contexts, such as model repair (Harel et al., 2012a; Katz, 2021); compositional and symbolic verification and analysis (Harel and Katz, 2014); automated optimization (Harel et al., 2013); synthesis (Greenyer et al., 2016); and in the context of deep learning (Yerushalmi et al., 2023; Ashrov and Katz, 2023).

Our agent is a scenario-based system constructed from declarative rules and domain-specific strategies. Some of these strategies were inspired by classical work in game-solving, most notably Allis' seminal thesis (Allis, 1988), which proves that Connect4 is a solved game where the first player can force a win using a combination of brute-force search and domain-specific heuristics. However, to our knowledge, no prior work has attempted to build a non-losing Connect4 player using only modular scenario-based rules combined with LLM-assisted development, nor to verify such an approach using formal methods.

9 CONCLUSION AND FUTURE WORK

Large Language Models (LLMs) have recently gained widespread popularity among developers for their ability to generate or assist in generating code. However, despite their utility, LLMs are often unreliable, difficult to verify, and struggle to produce correct behavior in complex systems. In this paper, we proposed a novel methodology that addresses these challenges by integrating LLMs with Scenario-Based Modeling (SBM). Our approach enables developers to leverage the strengths of LLMs — such as code generation and strategic reasoning — while mitigating their weaknesses through structured, modular, and verifiable design. By using SBM as a foundation, we facilitate an incremental development process in which LLMs contribute to individual scenario components rather than attempting to construct an entire system at once. This is enabled by SBM's modular architecture, where scenario objects operate independently. Such structure not only aligns well with the strengths of LLMs but also supports rigorous formal verification. It enhances the robustness and comprehensibility of the resulting code, making it easier to identify and correct errors. Ultimately, our approach demonstrates how LLM-assisted development can be carried out in a more controlled, reliable, and scalable manner. We demonstrated the effectiveness of the approach using the Connect4 game as a case-study.

Future research may focus on several key directions. One avenue is the development of more scalable and efficient model checking techniques tailored to SBM, enabling faster analysis of larger or more dynamic systems. Another is the application of our methodology to additional domains beyond games, such as robotics, smart environments, and human-computer interaction. Finally, we envision the creation of LLM agents specifically adapted for scenario-based reasoning — potentially by fine-tuning on SBM artifacts or incorporating behavioral constraints during generation — to further improve coherence, safety, and reusability within scenario-driven development.

ACKNOWLEDGEMENTS

This work was partially funded by the European Union (RobustifAI project, ID 101212818). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

Allis, V. (1988). A knowledge-based approach of connect-four — the game is solved: White wins. Master's thesis, Vrije Universiteit Amsterdam.

Allis, V. (2025). How to Win Connect 4. Technical Report. https://www.rd.com/article/how-to-win-connect-4/.

Anthropic (2023). Introducing Claude. https://www.anthropic.com/index/introducing-claude.

Ashrov, A. and Katz, G. (2023). Enhancing Deep Learning with Scenario-Based Override Rules: a Case Study. In *Proc. 11th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 253–268.

Berzack, A. and Katz, G. (2024). On Integrating Large Language Models and Scenario-Based Programming for Improving Software Reliability (Full Version). Technical Report. https://arxiv.org/abs/2509.09194.

Chowdhery, A., Narang, S., Devlin, J., et al. (2022). PaLM: Scaling Language Modeling with Pathways. Technical Report. https://arxiv.org/abs/2204.02311.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella,

A. (2002). NuSMV 2: An Opensource Tool for Symbolic Model Checking. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*.

Cursor (2023). Cursor: The IDE that helps you Code with AI. https://www.cursor.com.

Doe, J. (2025). Unbeatable ai — connect4. https://websim.com/@Ch13fB1gT4lk/connect-4-unbeatable-ai.

Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., and Lou, Y. (2024). Evaluating Large Language Models in Class-Level Code Generation. In *Proc. 46th Int. Conf. on Software Engineering (ICSE)*.

Galli, K. (2025). Connect4 Python GitHub Repository. https://github.com/KeithGalli.

Greenyer, J., Gritzner, D., Katz, G., and Marron, A. (2016). Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *Proc. 19th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 16–23.

Haque, M. (2025). LLMs: A Game-Changer for Software Engineers? Technical Report. https://arxiv.org/abs/2411.00932.

Harel, D., Kantor, A., and Katz, G. (2013). Relaxing Synchronization Constraints in Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 355–372.

Harel, D. and Katz, G. (2014). Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. In *Proc. 4th SPLASH Workshop on Programming based on Actors, Agents and Decentralized Control (AGERE!)*, pages 95–108.

Harel, D., Katz, G., Marron, A., and Szekely, S. (2024). On Augmenting Scenario-Based Modeling with Generative AI. In *Proc. 12th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 235–246.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2012a). Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 3–12.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2015). The Effect of Concurrent Programming Idioms on Verification. In *Proc. 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 363–369.

Harel, D., Lampert, R., Marron, A., and Weiss, G. (2011). Model-Checking Behavioral Programs. In *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pages 279–288.

Harel, D., Marron, A., and Weiss, G. (2012b). Behavioral Programming. *Communications of the ACM*, 55(7):90–100.

Katz, G. (2021). Towards Repairing Scenario-Based Models with Rich Events. In *Proc. 9th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 362–372.

Li, Y., Parsert, J., and Polgreen, E. (2024). Guiding Enumerative Program Synthesis with Large Language Models. In *Proc. 36th Int. Conf. on Computer Aided Verification (CAV)*.

Marron, A., Hacohen, Y., Harel, D., Mülder, A., and Terfloth, A. (2018). Embedding Scenario-based Modeling in Statecharts. In *Proc. 5th Int. Workshop on Model-driven Robot Software Engineering (MORSE)*, pages 443–452.

Priyanshu, A., Maurya, Y., and Hong, Z. (2024). AI Governance and Accountability: An Analysis of Anthropic's Claude. Technical Report. https://arxiv.org/abs/2407.01557.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. (2024). Clover: Closed-Loop Verifiable Code Generation. In *Proc. 1st Int. Symposium on AI Verification (SAIV)*, pages 134–155.

Tleemann, T. (2025). Pro Player — Connect4. https://tleemann.de/four.html.

Wang, S., Hu, M., Li, Q., Safari, M., and Yang, X. (2025). Capabilities of GPT-5 on Multimodal Medical Reasoning. Technical Report. https://arxiv.org/abs/2508.08224.

Yaacov, T. (2023). BPpy: Behavioral Programming in Python. *SoftwareX*, 24.

Yaacov, T., Elyasaf, A., and Weiss, G. (2024). Boosting LLM-Based Software Generation by Aligning Code with Requirements. In *Proc. IEEE 32nd Int. Requirements Engineering Conference Workshops (REW)*.

Yerushalmi, R., Amir, G., Elyasaf, A., Harel, D., Katz, G., and Marron, A. (2023). Enhancing Deep Reinforcement Learning with Scenario-Based Modeling. *Springer Nature Computer Science (SNCS)*, 4:1–13.