



# Enhancing Scenario-Based Modeling Using Large Language Models

David Harel<sup>1</sup>, Guy Katz<sup>2</sup>(✉), Assaf Marron<sup>1</sup>, and Smadar Szekely<sup>1</sup>

<sup>1</sup> The Weizmann Institute of Science, Rehovot, Israel

<sup>2</sup> The Hebrew University of Jerusalem, Jerusalem, Israel

`guykatz@cs.huji.ac.il`

**Abstract.** Manually modeling complex systems is a daunting task. Although numerous methods have been proposed for mitigating this issue, this difficult problem persists. Recent breakthroughs in generative AI and large language models have led to the creation of general-purpose chatbots, which can assist software engineers and modelers in various tasks. Still, these chatbots are often inaccurate or incorrect, and so using them in an unstructured manner might result in erroneous system models. Here, we outline a method designed for integrating chatbots into the modeling process, in a safer and more structured way. To facilitate this integration, we advocate the use of the scenario-based modeling paradigm, which has been shown to facilitate the automated analysis of models. We suggest that through the iterative invocation of a chatbot, combined with manual and automatic inspection of the models it produces, one can obtain a more robust and accurate system model. We report on favorable preliminary results, which showcase the potential of this approach.

**Keywords:** Large language models · Generative AI · Chatbots · Scenario-based modeling · Rule-based specifications

## 1 Introduction

The manual modeling of complex systems is an error-prone and daunting endeavor. Moreover, even after the initial modelling of the system is done, ongoing tasks such as repair and modification continuously tax human engineers. Creating methodologies and tools for facilitating and streamlining this process has been the topic of extensive research. Still, many aspects of this problem remain unsolved [5, 44].

In recent years, the deep learning revolution has brought about dramatic changes in many domains, including computer science. This trend has recently intensified with the release of ChatGPT, the learning-based chatbot [43], which appears to be a significant step towards general-purpose AI. ChatGPT, and other tools like it [13, 42], can be applied in countless kinds of tasks—and particularly, as part of the modeling and coding of complex software systems [45]. Using

ChatGPT, engineers might provide a natural-language description of the system being developed, and then receive from the chatbot a model of this system, or even the source code that implements it. Using iterative invocations of ChatGPT, the system can later be modified or enhanced, as the need arises. This kind of technique has already been successfully used on systems from various application domains [6, 39, 45].

Even though the ability to leverage ChatGPT<sup>1</sup> as part of the software development cycle can significantly empower software engineers, it also presents potential pitfalls that should be considered. One potential drawback of using chatbots in this way is that the answers they produce are often incomplete or inaccurate, and sometimes overlook various aspects of the input query [39]. Further, the input query presented to the chatbot might itself be imperfect, and the engineering team might overlook this fact until the system is deployed. Consequently, if we assume that human engineers will become increasingly dependent on chatbots for performing various tasks, the associated risk that such errors might find their way into the final, deployed models and code of the system at hand increases. Consequently, we face the following challenge: how can we harness modern chatbots in a way that lifts significant loads of work off the engineer’s shoulders, but which results in system models that are accurate and sound?

In this paper, we advocate for the design of an encompassing modeling scheme, which would allow engineers to combine chatbots such as ChatGPT with more manual, “traditional” techniques for systems modeling [5, 44], in such a way that will achieve the aforementioned goal. The core idea is to make use of ChatGPT in a more controlled way; i.e., by repeatedly invoking it for various tasks, but to also repeatedly analyze and inspect its results, in order to ensure their accuracy and soundness. We believe that such schemes, if properly designed, could allow software engineers to benefit from the capabilities of ChatGPT without jeopardizing the quality of the resulting systems. In the longer run, we argue that such a scheme could constitute a step towards the vision of *Wise Computing* [25], which includes the turning of the computer into a proactive member of the engineering team—which can propose possible courses of action, detect under-specified sections of the model, and support the various routine actions that arise as part of the development cycle of modern software.

To construct such a modeling scheme, we propose here to leverage some of the extensive work that has been carried out within the modeling community over the years. Specifically, we focus here on such modeling frameworks that afford two benefits which, in our view, complement the capabilities of ChatGPT: (i) models produced by the modeling framework are highly aligned with the way humans perceive the systems under development. This, we argue, could facilitate the manual inspection of ChatGPT’s output by the engineers; and (ii) models produced by the framework are amenable to automated analysis techniques, e.g.,

---

<sup>1</sup> We will often use the term *ChatGPT* somewhat generically, to represent an arbitrary, modern chatbot.

model checking. This is needed to support the automated detection of inconsistencies and errors in models that are automatically generated by ChatGPT.

Multiple existing modeling framework fit these requirements and could probably be used in our context. Still, for the initial evaluation results discussed here, we chose to focus on *scenario-based modeling (SBM)*. SBM is modeling technique that produces models comprised of simple *scenarios*. Each such scenario corresponds to a single aspect of the system being modeled [9, 33]. As we later discuss, this fact facilitates a smoother collaboration between human engineers and chatbot.

In order to demonstrate the potential of this combined framework, we study here some few tasks that naturally arise within a system’s life cycle. More concretely, we focus on the model’s initial design phase; its testing; the verification of its various properties; its later repair or enhancement, possibly due to the discovery of inconsistencies; and also the search for under-specified portions within the model. Our results are preliminary, but are highly promising. We hope that this paper could form the basis for further research in this important direction.

In the rest of the paper, we discuss the key concepts of our proposed approach, and lay out a high-level plan for future steps. We start with an introduction to the concepts of SBM and language model-based chatbots, in Sect. 2. We then present the proposed integration of ChatGPT and SBM in Sect. 3, and continue with a discussion of the more advanced aspects of such an integration in Sect. 4. In Sect. 5 we present technical and methodological challenges that emerged in our experiments as well as directions for addressing these challenges in further refinement of the methodology. Next, we discuss related work in Sect. 6, followed by a conclusion in Sect. 7.

## 2 Background

### 2.1 Large Language Model-Based Chatbots

Chat Generative Pre-trained Transformer (ChatGPT), by OpenAI [7, 43], is a large language model (LLM) based chatbot. ChatGPT is sufficiently powerful to conduct an iterative conversation of variable format, style, length, level of detail, and language. At each stage of the conversation, the user presents ChatGPT with a fresh prompt, and the chatbot then replies—based on all previous prompts that appeared in that conversation (also referred to as the *context*). Since its debut in 2022, ChatGPT has quickly become profoundly successful, and has inspired several other companies to create their own chatbots [3, 13, 42].

Internally, ChatGPT is comprised of a proprietary series of generative pre-trained transformer (GPT) models. These models are, in turn, based on Google’s transformer architecture [46]. ChatGPT has been fine-tuned for conversational applications, through a combination of reinforcement and supervised learning techniques, followed by manual adjustments by human engineers. The chatbot’s training, and also its inference, are considered very costly in terms of processing resources and power consumption.

As far as functionality is concerned, ChatGPT is extremely versatile. Some of its numerous uses include writing and debugging computer programs [45], generating student essays [2], and also composing music [40]. However, it may sometimes produce plausible-sounding but completely incorrect answers—which is a known limitation for large language models [16].

## 2.2 Scenario-Based Modeling

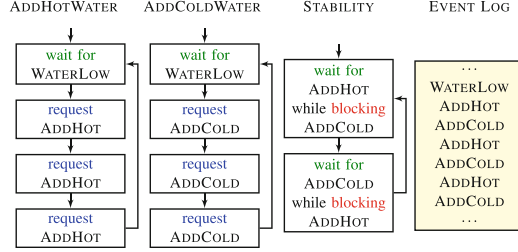
Scenario-based modeling [33] (SBM) is an approach for modeling complex, reactive systems. The main building block in a scenario-based (SB) model is called a *scenario object*, and it describes a single behavior of the system being modeled—whether desirable or undesirable—which can be specified as allowed, necessarily, or forbidden. Each of the model’s scenario object does not interact with its counterparts directly, and is created in isolation. Cross-scenario interaction is supported, but only through a global execution mechanism, which may execute a set of scenarios in a way that produces global, cohesive behavior.

Multiple flavors of SBM have been proposed, each with its own mechanism for cross-scenario interactions. Here, we focus on a particular set of idioms, which are common in many of the SBM frameworks: the *requesting*, *waiting-for* and *blocking* of discrete event sets [33]. As the mode is executed, each of the scenario objects repeatedly visits designated *synchronization points*; and in each of these points, the global execution mechanism triggers a single event. Each scenario object may declare sets of events that it wishes to see triggered (*requested* events), events that it would like to avoid (*blocked* events), and events that it does not request itself, but which it would like to monitor (*waited-for events*). The global execution mechanism then collects these declarations from each of the scenario objects in the system (or, possibly, a subset thereof [19]), selects a single event that is requested and not blocked, and then broadcasts this selection to all relevant scenario objects.

In each synchronization point there may be multiple events that are requested and not blocked; and multiple strategies have been put forth for selecting one of these events. Possible strategies include arbitrary or random event selection, a round-robin mechanism, and also look-ahead schemes that simulate the possible progression of the execution, and then select events in order to optimize an objective specified a-priori (e.g., the avoidance of deadlocks). Executing a scenario-based model in this way is termed play-out [31]).

Figure 1 illustrates a simple SB model. It represents a system that controls the water level of a water tank, which is equipped with cold and hot water taps. Each of the scenario objects is depicted as a transition system, wherein nodes correspond to the predetermined points of synchronization. The ADDHOTWATER scenario object repeatedly waits for events of type WATERLOW, and whenever such an event is triggered, the scenario object requests three times the event ADDHOT. Symmetrically, the ADDCOLDWATER scenario object requests the addition of cold water. When the SB model includes only the objects ADDCOLDWATER and ADDHOTWATER, three ADDCOLD events and three ADDHOT

events may be triggered, in any order, during the model’s execution. If maintaining a more stable water temperature within the tank is important, this could be achieved by adding the scenario object STABILITY, which enforces the interleaving of ADDHOT and ADDCOLD events—through the use of the event blocking idiom. An execution trace of the SB model that contains all three objects appears in the event log.



**Fig. 1.** (Borrowed from [26, 28].) An SB model that controls the water level in a tank with hot and cold water taps.

Scenario-based modeling has been implemented on top of several high-level languages, such as C++ [21], Java [32], JavaScript [4], Python [47], and ScenarioTools [14]. Moreover, scenario-based modeling has been applied to various complex systems, including cache coherence protocols [23], a web-server [21], and robotic controllers [17]. To keep the presentation simple in the coming sections, we will usually describe SB models as transitions systems.

We follow here the formal definitions of SBM as they appear in [36]. A scenario object  $O$  over event set  $E$  is given as a tuple  $O = \langle Q, \delta, q_0, R, B \rangle$ , where the components are defined as follows:

- $Q$  is a set of states. Each state in  $Q$  represents a single, predetermined synchronization point;
- $q_0 \in Q$  is the initial state;
- $R : Q \rightarrow 2^E$  and  $B : Q \rightarrow 2^E$  are mappings from states to the respective sets of events requested and blocked at these states; and
- $\delta : Q \times E \rightarrow 2^Q$  is a transition function, which serves to indicate how the scenario object transitions between states in response to the triggering of events from  $E$ .

A set of individual scenario objects can be composed, in a pairwise fashion, according to the rules that appear below. Let  $O^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1 \rangle$  and  $O^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2 \rangle$  be two scenario objects, specified over a common event set  $E$ . These two objects are composed into a single scenario object  $O^1 \parallel O^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2 \rangle, R^1 \cup R^2, B^1 \cup B^2 \rangle$ , where:

- $\langle \tilde{q}^1, \tilde{q}^2 \rangle \in \delta(\langle q^1, q^2 \rangle, e)$  if and only if  $\tilde{q}^1 \in \delta^1(q^1, e)$  and  $\tilde{q}^2 \in \delta^2(q^2, e)$ ; and

- the composite labeling function is defined as the union of its constituents; i.e.,  $e \in (R^1 \cup R^2)(\langle q^1, q^2 \rangle)$  if and only if  $e \in R^1(q^1) \cup R^2(q^2)$ , and  $e \in (B^1 \cup B^2)(\langle q^1, q^2 \rangle)$  if and only if  $e \in B^1(q^1) \cup B^2(q^2)$ .

Through the  $\parallel$  composition operator, we define a *behavioral model*  $M$  as a set of scenario objects,  $M = \{O^1, O^2, \dots, O^n\}$ . The executions of model  $M$  are defined to be precisely the executions of the composite object  $O = O^1 \parallel O^2 \parallel \dots \parallel O^n$ . Consequently, each execution of  $M$  starts from  $O$ 's initial state, which, in turn, is the  $n$ -tuple of initial states of the individual objects of which  $O$  is comprised; and throughout  $M$ 's run, in each state  $q$  one enabled event  $e \in R(q) - B(q)$  is selected for triggering, if such an event exists. The execution then transitions to a state  $\tilde{q} \in \delta(q, e)$ , and the process is repeated.

### 3 Integrating ChatGPT and SBM

#### 3.1 Basic Integration

As the first step towards integrating SBM and ChatGPT, we present a straightforward methodology for generating scenario objects from free-text, using ChatGPT. To get ChatGPT to present its output in scenario object form, we propose to include, in the prompts presented to the chatbot, a *preamble* that describes SBM's semantics, as presented in Sect. 2. This preamble is used to instruct ChatGPT to produce its output as transition systems, wherein each state is labeled with requested/blocked events; and has also a set of transition rules that indicate its successor state, for each event that could be triggered. Note that such a preamble only needs to be designed once, and that it can then be used with any subsequent query. Whenever a specific SBM implementation is in use (e.g., C++, Java), the chatbot's output can be adjusted to match the required syntax of that implementation.

The simple preamble, which we used in our some of our initial experimentation, is as follows:<sup>2</sup>

I would like you to help me create a scenario-based model. A scenario-based model is comprised of a finite set of events, and scenarios. Each scenario is a sequence of states. In each state the scenario can request events, wait-for events, and block events. Afterwards, one event that is requested by at least one scenario, but is blocked by no scenario, is selected for triggering. Scenarios that requested or waited-for this event may then change states, and the process repeats itself. Scenarios cannot have internal loops or variables; these must be expressed using additional states.

To display scenarios, please use the following format. For each state, mention the state's name, and the sets of requested and blocked events. Also, mention the transitions to other states. For example:

- s1: request X, block Y. If X is triggered, go to state s2.

<sup>2</sup> This preamble and several other prompts are borrowed from [26]. In some of the experiments, we used other descriptions of SBM; but as it turned out, ChatGPT was able to handle very well the great variability in such descriptions.

- s2: request Y, block X. If Y is triggered, go to state s2.

Now, here are the details of the specific scenario-based model I'd like you to create...

Through this preamble, we could get ChatGPT to generate the model we described in Sect. 2. For the ADDHOTWATER scenario, we used the following prompt:

Consider the event set WaterLow, AddHot, AddCold.

Please suggest a scenario that waits for event WaterLow (without requesting it). When WaterLow is triggered, the scenario performs the following, 3 times in a row: request the event AddHot, and wait for it to be triggered (both in a single state). When the 3rd AddHot event occurs, the scenario transitions back to its initial state.

The chatbot then replied to this prompt as follows:

- s1: wait for WaterLow. If WaterLow is triggered, go to state s2.
- s2: request AddHot, wait for AddHot. If AddHot is triggered, go to state s3.
- s3: request AddHot, wait for AddHot. If AddHot is triggered, go to state s4.
- s4: request AddHot, wait for AddHot. If AddHot is triggered, go to state s1.

Which is indeed equivalent to the transition system that appeared in Fig. 1. The ADDCOLDWATER scenario could also, be generated using the symmetrical prompt. Next, if we wished to incrementally modify the system through the introduction of the STABILITY scenario, we could issue the final prompt:

Please suggests a scenario that uses blocking to ensure that no two consecutive AddHot events can be triggered, and that no two consecutive AddCold events can be triggered; that is, once AddHot is triggered, AddCold must be triggered before AddHot can be triggered again, and vice versa. This scenario should not request any events, and should work regardless of any WaterLow events.

And in response, ChatGPT would produce the STABILITY scenario, precisely as described in Fig. 1.

We point out that there is a subtle difference between the prompts we used to obtain the first two scenarios, ADDHOTWATER and ADDCOLDWATER, compared to the prompt used to obtain STABILITY. For the first two cases, our prompt included information that approximately described the transition system that we desired; whereas in the last case, our prompt was more high-level, and did not contain the word “state”. Still, in all cases, the chatbot successfully produced the desired outcome. This further demonstrates the wide specification range that ChatGPT can successfully handle; and it suggests that a chatbot can

be used even in cases where the engineers themselves are not entirely certain of what the scenario that they desire looks like. While it is reasonable to assume that the more accurate description, the more accurate result, it seems that even high-level descriptions can prove useful, especially when enhanced by techniques for automated analysis, as we discuss next.

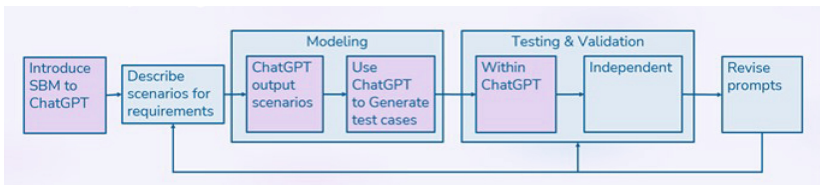
### 3.2 The Proposed Methodology

Further extending the basic integration between SBM and ChatGPT, we now propose an outline for a structured, language-agnostic and LLM-agnostic methodology for creating complex models of reactive systems that interact with their environment repeatedly, and may receive external inputs [35]. Many critical, modern systems can be regarded as reactive [1], and as a result there has been extensive work on devising methods and tools for modeling such systems. In spite of this tremendous effort, there still remain significant gaps; and these could result in models that are either inaccurate or difficult to maintain, or both. In this paper, which can be regarded as an element within the Wise Computing vision [25], we seek to mitigate these gaps, by creating advanced and intelligent tools, which will begin to undertake system development tasks that are traditionally reserved for humans. The approach is based on having system components generated, incrementally and iteratively, through the use of an LLM; and to have the LLM's outputs checked systematically, and semi-automatically, using various methods and tools (see Fig. 2).

1. Describe, textually and in natural language, the problem and its environment.
2. Choose a scenario-based, compositional modeling language, which affords well-defined execution semantics, and which is suitable for incrementally developing the system at hand.
3. Obtain a chatbot that is familiar with the general application domain, or which can readily become familiar with the domain through external knowledge, and which can be made to produce code in the selected scenario-based language.
4. Create a preamble that iteratively describes the semantics of the scenario-based language to the LLM. In order to confirm that the LLM has successfully internalized the details of the language semantics, have it execute (i.e., play out [31]) systems described as rules or scenarios in the selected language. Specifically, have the LLM output logs of scenario states, triggered events, composite system states, the values of environment variables, etc.
5. In an iterative manner, add scenarios and refine existing ones, as follows:
  - (a) Use prompts to describe certain not-yet-specified requirements or aspects of the system as scenarios.
  - (b) Have the chatbot generate the actual scenarios for the prompt, in the selected language.
  - (c) Have the chatbot generate natural language descriptions of executable test cases and properties to be verified (perhaps as assertions for formal verification tools), per the original requirements. This entails stating the requirements at hand, from different perspectives.



- (d) Carry out initial validation and testing within the chatbot, by challenging it to independently find gaps and incorrect execution paths. If needed, correct the natural language specification and prompts.
- (e) Systematically check the output produced by the LLM, externally, by using the following (or part thereof): unit testing of individual scenarios, code reviews by human engineers, model checking of the new scenarios, as well as those of the composite system, subsystem testing with some or all of the already-developed scenarios, etc. Such testing is intended to be carried out in the execution environment of the language, whereas model checking is intended to be carried out using an adequate formal verification tool. Most importantly, both should be independent of the LLM's environment. As a possible enhancement, automate the subjecting of generated scenarios to model checking and testing.
- (f) When errors are discovered, do not alter the generated code; instead, alter the LLM prompts, until the correct system scenarios and testing and verification properties are successfully generated. This step is crucial for ensuring that the stakeholder's (i.e., customer's) view of the requirements is well aligned with the developer's understanding, as well as with the actual code.
- (g) Once the set of generated scenarios appears ready, repeat step (d), by having the LLM find potential failures or gaps in this set of scenarios. Specifically, attempt to find, through the LLM's suggestions, new environment considerations that might prevent the system from functioning correctly. This step is intended to mimic common system engineering task of having potential customers or external experts review some of the advanced system prototypes. Then, repeat earlier steps as needed.



**Fig. 2.** Methodology overview: Development is iterative; testing and validation are carried out both within and outside of the LLM; when issues are identified with LLM-generated code, problematic code blocks are replaced using new or revised prompts.

Next, we elaborate on some of these steps, and provide illustrative, simple examples.

## 4 Using the Method in the Development Cycle

### 4.1 Code Generation

Code generation is one of the most straightforward capabilities of chatbots that we propose to integrate into a system’s development cycle. In Sect. 3 we demonstrated that ChatGPT can generate an executable, scenario-based model—and similar capabilities have been demonstrated with other languages [6, 39, 45]. One advantage in the context of scenario-based systems is that it is possible to generate stand-alone scenarios, which can then be tested and reviewed separately, and later be added, incrementally, to the system at hand. In our preliminary experimentation for this paper, we tested code generation for requirements in the realms of algorithms on data structures, autonomous vehicles, control systems, and simulating natural phenomena. For each of these realms, the chatbot/SBM integration proved useful.

### 4.2 Modeling

Once ChatGPT successfully digested the underlying principles of SB models, it was indeed successful in combining logic with its knowledge of the problem domain and of the world at large, in order to enhance or develop an SB model. For example, it was successful in introducing new environment events, in describing the sensor scenarios required for triggering such events, and also in adding the corresponding application scenarios that react to these events. In one case, we considered a robot moving numbered tiles on a sequence of cells, and asked the chatbot to generate SB code for a bubble-sort algorithm to be used by this system. ChatGPT was able to introduce events corresponding to the arrival of a tile at the tail-end of the array, and it was also able to generate scenarios for reacting to such events.

### 4.3 Play Out and Simulation

After a few attempts, we succeeded in getting ChatGPT to produce execution logs for arbitrary SB specifications. Initially, we observed “wishful thinking”, where the chatbot would describe an execution as it should be per the problem description. By iteratively refining our prompts, as illustrated in Fig. 3, we eventually got ChatGPT to follow the execution steps correctly and to display the event and transition leading to each synchronization point; as well as a table of all scenarios, indicating for each scenario its event declarations and whether or not it had requested or waited-for the event that was eventually triggered.

### 4.4 SMT-Like Model Analysis

A key advantage of SB modeling is its amenability to automated formal verification, and the availability of tools that perform this—either through exhaustive model checking that traverses all paths, or by using a domain-specific

**Scenario descriptions in user prompts to ChatGPT:**

GL: Repeat three times: request the event "Flash Green Light"  
 RL: Repeat three times: request the event "Flash Red Light"  
 ALTG: Repeat forever : Wait for "Flash Green Light" and after it happens,  
       block "Flash Green Light" while waiting for "Flash Red Light"  
 ALTR: Repeat forever : Wait for "Flash Red Light" and after it happens,  
       block "Flash Red Light" while waiting for "Flash Red Light"

**Excerpts from ChatGPT payout output:**

Upon initialization, all scenarios are allowed to reach their first stable state.

Scenario Short name	Wake-up	Requested	Blocked	Waited for
GL	Y	Flash Green Light		
RL	Y	Flash Red Light		
ALTG	Y			Flash Green Light
ALTR	Y			Flash Red Light

Let's proceed with the next step following the event "Flash Green Light":

Scenario Short name	Wake-up	Requested	Blocked	Waited for
GL	Y	Flash Green Light		
RL	N	Flash Red Light		
ALTG	Y		Flash Green Light	Flash Red Light
ALTR	N			Flash Red Light

**Fig. 3.** (Borrowed from [26].) *Simulating the execution of an SB model with ChatGPT.* Top: user prompts describing the scenarios at hand. Middle and bottom: The first couple of entries in the chatbot's output, when requested to play out the system's execution. These show the triggered events, the scenarios that reacted to these events, and the event-set declarations of each of the scenarios at the following synchronization point.

verification approach through the use of Satisfiability Modulo Theory (SMT) engines [20, 30, 37]. This is achieved through the abstraction and encapsulation of domain-specific processes, conditions and actions as states and events. The result is a set of relatively small intuitive scenarios that reflect individual requirements, and whose composition produces the complex system—as opposed to complexities that rise from the intricate conditional flow of sensitive and delicate processes, each with numerous steps.

In our experiments, we observed that ChatGPT is able to leverage these kinds of encapsulation and abstraction in identifying cases where a specification has either been omitted, or is handled incorrectly. In one case involving an autonomous vehicle system, we presented the chatbot with the following three requirements: (i) always stop when the light is red; (ii) always obey instructions from a police person; and (iii) never injure a person. ChatGPT readily reported that these requirements may be conflicting. With regards to the safety requirement of not entering an intersection when the traffic light is red, the chatbot pointed out that the autonomous vehicle may fail to stop if the road is icy; that it may involuntarily enter the intersection if, after stopping, it is hit from behind by another vehicle that did not stop; and, furthermore, that it may cause injury

to a person even without moving, e.g., if a person who is walking behind the AV gets hit by another vehicle and is thrust against the AV.

In another case, using a mathematical, SMT-like analysis, ChatGPT successfully identified a particularly interesting execution path. Here, we presented the chatbot with a system comprised of the following four scenarios: the first scenario waits for an external time tick event, then requests flashing a red light briefly, and then waits for any event; the second scenario does the same, but requests to flash a green light; the third scenario counts time ticks modulo 3, and blocks the red light from flashing every third tick; and the fourth scenario another blocks flashing the green light, every five ticks. Next, we added the requirement that between any two consecutive time ticks, at least one light must be flashed. When prompted, ChatGPT responded that the requirement is not satisfied—and that every 15 time ticks ( $3 \cdot 5$ ) both red and green flash events will be blocked. The chatbot apparently reached this conclusion through an arithmetic calculation, as opposed to a step-by-step simulation, which allowed it to quickly reach similar conclusions for arbitrarily large numbers.

Below is an excerpt from this conversation (borrowed from [26]), after the scenarios had already been described. When prompted with

now a new requirement came in: always, after every tick, *at least one light* must be flashed. Does the combined system of red flashing and green flashing with the 3 and 5 restrictions satisfy the requirement?

To this, ChatGPT replies:

To analyze whether the combined system satisfies the new requirement that at least one light must flash after every tick, we must consider the synchronization points of both the red and green systems with their respective restrictions:

The red system flashes on every tick, except every third one. The green system flashes on every tick, except every fifth one. If we consider the sequence of ticks and apply both restrictions, the combined system would not flash a light on a tick that is both a multiple of three and five — specifically, on ticks that are multiples of fifteen, since 15 is the least common multiple of 3 and 5. At such ticks, both systems would be blocking their respective lights from flashing due to their individual restrictions.

Therefore, on ticks that are multiples of fifteen (like the 15th, 30th, 45th tick, and so on), neither light would flash, violating the new requirement that at least one light must flash after every tick.

In conclusion, the combined system of red flashing and green flashing with the restrictions of not flashing on ticks divisible by three and five, respectively, does not satisfy the requirement that at least one light must be flashed after every tick.

Further, the chatbot was successful in discussing possible generalizations of this computation, including the calculation of the number of states of the composite system that an exhaustive model checker would have to traverse.

However, this capability is limited and should be used with care. For example, when we presented a similar case, where event E1 occurs every 3s, in seconds 0,3,6,9, etc., and event E2 occurs at second 77 and then every 75s, it took several interactions and correction by humans for ChatGPT to realize that E1 and E2 can never occur at the same second since the time stamp of E2 events is always 2 modulo 3. This deficiency does not completely erase the value of ChatGPT in such SMT-like analyses. When checking global assertions, counter examples yielded by ChatGPT can be readily checked, and may identify potential bugs. Incorrect counterexamples may highlight gaps in the specification which may have led to the incorrect computations. Plain errors by ChatGPT in this context are not too expensive. Partial or incorrect attempts by ChatGPT at proofs that certain properties always hold, can help guide humans in carrying out rigorous analysis, whether mentally or with the help of tools, independent of an LLM.

Interestingly enough, while the chatbot successfully applied its logic to discover system execution paths with special properties, it failed to perform as well when required to list *all* possible execution paths that possess a certain property. In one case, we presented ChatGPT with a 4-by-4 cell grid, and prompted it for the steps that a robot may take from the bottom-left cell (0,0) to the top-right cell (3,3), using the usual events for up, down, left and right motions. We also instructed the chatbot to only look for paths that remained within the grid. ChatGPT produced paths such as *right, right, right, up, up, up* or *right, up, right, up, right, up*, but when instructed to list all possible paths that go through cell (1,3) it failed, and instead listed several paths that did not reach the desired target at all, such as *up, up, up, right, down, down, right, up, right*. We thus conclude that ChatGPT is not yet able to carry out exhaustive model checking on SB systems. However, this is not a severe limitation, because our proposed methodology for verifying the LLM’s outputs entails carrying out such analysis independently of the LLM environment.

## 4.5 Identifying Under-Specification

Even in cases where an existing model addresses all stated requirements perfectly, it is possible that there are additional areas of the broader problem domain that are currently not handled properly—possible due to synthetic or reduced problem descriptions. When presented with appropriate prompts, ChatGPT was able to significantly improve an already developed model, or, conversely, to help in explicitly stating the expected scope and boundaries of the intended system.

In one example [18], we presented the chatbot with an SB model that had been developed for science teaching purposes. The model contained information on how bees depend on the ambient temperature in order to switch between being active and hibernating; how flower plants rely on hours of daylight in order to start blooming; and how active bees depend on blooming flowers for their

survival. In the original model, the number of daylight hours and the average temperatures were mutable parameters, which students could adjust in order to check how certain combinations of settings could sustain the bees, whereas other settings could cause the bees to die out due to hunger.

When we prompted ChatGPT for possible enhancements to the education model, it returned with a suggestion that we consider also weather-related variations, as these might modify the pre-specified fixed number of daylight hours and average temperature for a given time of year.

#### 4.6 Explainability

Once the chatbot is provided with a certain expected or observed behavior of the system, it is often able to explain the cause of that behavior, in natural language. Further, it is often able to reasonably balance between providing the overly detailed, complete execution trace, and the overly simplistic explanation that “this is what the system is supposed to do”.

In one case we provided the chatbot with details on the semantics of SB modeling, and then prompted it as follows (borrowed from [26]):

In a system with two scenarios, one waits for time tick events and after each time tick requests a short beep (100 ms). The other scenario waits for time ticks, and after each such event requests a long beep (500 ms). Each of these scenarios waits for both the short and long beep events, so that after each time tick, exactly one beep occurs. Actuator scenarios actually cause the respective beeping action. Time tick events are injected by the environment every one second. A user reported that it watched the system for 20 s, and never heard a long beep. Can you explain why this may have happened?

ChatGPT’s response was:

The user’s report of not hearing a long beep in a system with two scenarios—one requesting a short beep after each time tick, and the other requesting a long beep after each time tick—can be explained through the principles of Scenario-based Modeling (SBM).

In SBM, when multiple scenarios request different events after a shared trigger event, the execution environment selects one of these requested events to occur. The selection can be arbitrary or follow a specific strategy, such as alphabetic or lexical order, priority, random, round-robin, etc. In this case, if the event selection strategy prioritizes the short beep over the long beep (for example, due to alphabetic order, where “short beep” comes before “long beep”), then the short beep will always be selected to occur after each time tick, and the long beep will never be executed. Since time tick events are injected by the environment every second, and both scenarios are waiting for this event, each time tick leads to both scenarios being activated. However, if the system consistently selects the short beep

event over the long beep event after each time tick, the scenario requesting the long beep will never progress to the state where it actually triggers the long beep. As a result, only short beeps will be heard.

In summary, the user's observation of not hearing any long beeps can be attributed to the event selection mechanism within the SBM framework, which in this case consistently favors the short beep event over the long beep event. This could be due to the selection strategy used by the system (like alphabetic order) or other operational logic specifics.

We then challenged the chatbot again:

Thank you. Assume that the developers changed the selection algorithm and they now select events randomly. The user tested the system many times, for hours, and never heard a long beep. Can you try to explain?

And in response to this, ChatGPT returned with a highly comprehensive answer, which covered many possible issues with the execution infrastructure, human-machine interactions, and the application scenarios. Here are a few excerpts, borrowed from [26]:

If the event selection algorithm was changed to randomly select between the short beep and long beep events, but the user still never hears a long beep, there are several potential explanations:

**Implementation Error in Random Selection:** The most straightforward explanation could be a bug or implementation error in the random selection algorithm. It might not be truly random, or it could be incorrectly favoring the short beep event due to a coding mistake.

**Issue with the Long Beep Scenario:** There might be a problem with the scenario that requests the long beep. This could be a coding error or a logical flaw in the scenario itself, preventing it from properly requesting the long beep event.

**User Perception or Reporting Error:** Finally, there could be an issue with how the user is perceiving or reporting the beeps. They might be missing the long beep due to distraction, ambient noise, or misunderstanding what constitutes a long beep in the context of this system.

To diagnose the issue accurately, developers would need to conduct thorough testing and debugging, looking into each of these potential areas to find where the problem lies.

Although these explanations were quite impressive, they were also imperfect. In some cases, we presented the chatbot with well-specified scenarios accompanied by a detailed description of an observed event sequence, but ChatGPT was unable to describe a step-by-step execution (play-out) of the scenarios that would result the given event sequence; instead, it provided incorrect explanations for how the event sequence might occur. Such issues could sometimes be resolved by repeatedly providing ChatGPT with reminders and pin-pointing its

mistakes. This is a significant caveat that serves as a reminder that outputs produced by the LLM must be formally inspected and verified.

#### 4.7 Accommodating Semantic Flexibility

Many software engineering teams are tied to specific programming and modeling languages, and their associated semantics. In our experimentation, ChatGPT was in many cases able to discuss and accommodate alternative semantics, demonstrating flexibility that human engineers sometimes lack.

For example, consider again the water tap example, first described in Sect. 2. Observe the ADDHOTWATER scenario: when it is in any of the states where it requests ADDHOT, it is not waiting for event WATERLOW, and consequently cannot react to its triggering. However, if we consider the semantics of the Live Sequence Charts [9], which is another SB formalism, every scenario constantly waits for all events that were waited-for in its starting state. When such a waited-for event occurs, the LSC infrastructure instantiates a fresh copy of the scenario. As it turns out, from our initial textual description of SBM, ChatGPT concluded that this semantics were the default.

In another example, we tried to get ChatGPT to generate a scenario-based implementation of Quicksort. Before starting, the chatbot remarked that this will be hard, because classical solutions are recursive. To assist the chatbot, we then pointed out that there was actually an alternative, published implementation for Quicksort, which was iterative instead of recursive [34]. Further, this implementation was structured as a set of instructions to human workers that needed to arrange cars in an automobile dealership parking lot, according to window-sticker prices. Each employee in this implementation was assigned one, narrow role, which coincides nicely with how SB modeling works. ChatGPT readily accepted this new “mindset”, and was able to produce the sought-after SB specification.

#### 4.8 Interactive Mutual Learning

Through our experimentation with the proposed approach, we observed that ChatGPT could often achieve better results when it was provided with multiple prompts, explorations and discussions, compared to when it was provided with a single, concise, detailed description. We believe that this might be a general rule worth following when a chatbot is used as part of the software and system development cycle. Thus, this style of work might be a good fit for an agile development process, which inherently involves a great deal of trial and error as it spirally converges towards the specified (or, sometimes, under-specified) goal. From the chatbot’s point of view, this is a highly constructive process, in which developers and stakeholders iteratively describe their wishes and plans, as they slowly refine their own understanding of the system and its environment, as well as their own needs and intended future interactions with the system.

An integral part of this refinement is the production of increasingly explicit definitions of elements that are within the scope of the system’s environment.



Such definitions, which may be highly useful to both chatbots and humans, are often quite absent from classical system specifications.

#### 4.9 Generating New Formal Assertions and Testable Properties

Given an LLM-generated component, generating test cases based on the original specification is a straightforward activity. Furthermore, it is only natural to ask the LLM to generate the code for such a test case. However, the generated programs, or scenarios, must be tested also from additional perspectives. Specification and execution of these tests may be carried out both with and without the help of the LLM.

To illustrate these additional perspectives, consider the following examples.

We have asked ChatGPT to produce a stand-alone Python program that flashes a green light for half a second every 3 s, i.e., in seconds 0, 3, 6, 9, etc., when counting time from zero. ChatGPT returned an acceptable program. When asked to generate a test case, it created a reasonable program that examined an execution log and confirmed that indeed one would observe repetition of periods of half a second of the green light being on, followed by 2.5 s of that light being off. However, as may often happen, if the on and off periods are not exactly 0.500 and 2.500 respectively, but say, 0.501 and 2.501, the program may still pass the test, but over time, the timestamps of when the light is flashed will be quite far from a multiple of 3. This is another property that must be checked as it was in the specification. Generating the code for this test may be done by a human or by ChatGPT (of course, the test case must be checked in either case).

One technique for generating new perspectives is to generate alternative designs and implementations, and examining the differences. We have asked ChatGPT to generate code for a robot to navigate a two dimensional grid of cells from an origin cell to a destination cell with steps of right, left, up and down, and where some cells are marked as obstacles, and can be sensed only from an adjacent cell. We then asked ChatGPT to generate a materially different program. ChatGPT generated a breadth-first search, replacing its choice of depth-first search in the first program. When asked about the externally visible effects it listed resource usage and performance aspects, opportunity for parallelization, and more. We then clarified to ChatGPT that the grid data structure is not to be used as a map in advance route planning, and that the scenarios should simulate actual movement, including back-tracking. While such simulation seems very similar to a depth first search, the path that the program outputs would be different: in the simulated traversal, the backtracking steps are part of the path, where in the search they are only part of the planning until a path with no backtracking is found. A developer can examine these issues and determine whether any of them should be included in the specification or in the testing or formal verification of the component.

Another approach for generating new perspectives does not involve code generation, but explores possible behaviors. Going back to the light flashing program from the first example in this subsection, we asked ChatGPT whether a program could comply literally with every aspect of the specification, and still produce

undesired results. In its first response, ChatGPT suggested that a program that flashes a red light between consecutive flashing of the green light may comply literally with the specification, but may not behave as expected. Similarly ChatGPT suggested that if the device producing the green light also creates a sound, the behavior may be undesired. The developers can then consider whether to enhance the specification accordingly to avoid these cases, and/or whether to create test cases, using cameras, microphones, and appropriate software, to check that indeed the behavior of the system in the world is as intended.

The properties to be checked can be formulated by the LLM in natural language, as executable test cases, or as we show below, as formal assertions for verification tools. For example, we prompted ChatGPT to analyze the following scenario: An autonomous vehicle (the ego car) is at a red traffic light. It senses that car B, that was approaching fast behind it, noticed the red light late, and hit its brakes. B is screeching to a halt. It is not clear whether B will be able to stop before it hits the ego car. Further, parts of the intersection are “clear”. With some interaction, ChatGPT produced reactive code for the ego car to move forward when both needed and possible. Clearly, it is not enough that the ego car should eventually reach a point beyond the stopping point of car B, and we asked ChatGPT to refine and formalize this requirement. Referring to variables in the program, it offered two variants, one for SPIN/Promela:

$$ltl \text{ no\_collision}\{[]((d\_E - d\_B) \geq \text{safe\_distance})\}$$

and one for NuSMV:

$$\text{SPEC AG}((d\_E - d\_B) \geq \text{safe\_distance})$$

## 5 Methodology and Technology Notes

In this section we document several issues and challenges that came up in our experiments in using an LLM according to the methodology outline described above. For each issue we list possible approaches for addressing it in further formalization or refinement of the methodology, in developing tools around it, and/or in the way engineers apply the methodology. The discussion refers to experiments with development of a scenario-based program for grid traversal.

### 5.1 The Methodology Focuses on One Aspect of the Development Cycle

The use of an LLM in building software applications is already wide-spread, and will likely grow, in analysis, requirement specification, design, coding, testing, etc. The methodology discussed here focuses particularly on the *development* task (which is largely code development), as contrasted, say, with requirement specification or integration testing; it can be employed within an agile-development sprint, and within other approaches to the system development cycle.

## 5.2 Prompt Engineering for Scenario Generation

What should a prompt that requests an LLM to generate one scenario look like?

Ideally, one would like to specify the desired function in a natural manner, as in “Do not allow the robot to move into a grid cell that is marked as an obstacle”, and let the LLM take care of the details of SBM. Alternatively, one could specify “Please create a scenario that, before the robot moves out of a grid cell, checks the cells to the right, left, above and below the current cell, and if any of those adjacent cells is an obstacle, block the respective, Right, Left, Up, and Down move events, until the robot exits the current cell.”. However, if the LLM is able to add fine details to the actual code, it means that the code is more distant from the requirement specification, and may contain decisions and choices that a domain specialist may not be aware of. For example, following multiple prompts, the LLM may make application-related assumptions (as in the case of carrying out a map-based search instead simulated traversal as described in Sect. 4.9). Of course, such LLM decisions would turn out in code review, and then either be accepted or rejected by the developer. Clearly, in a methodology that aims for intuitive executable specifications, the natural language prompts should be as close as possible to the actual code. Hence, when the LLM indeed enhances the generated module beyond the original prompt specification, the developer should update the prompt to either request or exclude these enhancements specifically.

One could also start with a functional description of a scenario, and once this is complete, after several iterations, have the LLM generate, in addition to the SBM code, a natural language description of the scenario. In addition to serving as documentation, the description may serve as an LLM prompt in another scenario-generation iteration (which of course must be tested).

## 5.3 Watching Out for Scenario Regression

The output of an LLM is never predictable. Small and focused changes in a prompt, or new prompts requesting a small change in an existing code block may cause larger, or unpredictable, changes in one or more scenarios. Our conclusion is that with each iteration of producing one or more artifacts one should (a) compare the result with prior artifacts; and (b) run tests and a comprehensive manual (i.e., human-driven) review over a larger set of artifacts and components. Tools that readily compare and display all differences between multiple LLM-generated code blocks would be very useful for this purpose. It is not enough to observe the desired change in the expected location in the code, or the changed behavior in the problematic section in an execution log. While this is an intuitive methodological requirement and common practice, recall that the entire methodology is about the coding stages in application development and not about application testing stages.

## 5.4 Dealing with Context Dependencies

As stated earlier, scenarios implicitly start with the word “always”, as in: “Always when event E1 occurs, request event E2”. However, this universal rule may have

exceptions that depend on the context, i.e. a variety of system and environment conditions. For example, when hurrying to a hospital, transporting a patient in a critical condition, on a deserted street, when no other vehicle is in sight, a human driver may (cautiously) run a red light, and an autonomous, SBM-based vehicle might need to do the same. Or, in our grid traversal example, the robot creates its own context, and the path should not loop into itself. Should the necessary information be tracked in each scenario separately? Should there be a data structure that is shared between the scenarios? Or, should the environment inject external sensor events to indicate relevant information?

There has been research about context oriented programming in general [8], and implementing context awareness in behavioral programs in particular [11]. In the present example, we chose to have several shared data structures for this purpose, but the more general question of what are the best ways to weave context dependencies SB designs, and therefore into LLM prompts, is still open.

## 5.5 Dealing with LLM-Generated Bugs

The LLM may generate incorrect code even for the most precise prompt. Here are a few examples from our experiments: (i) when checking certain properties of adjacent cells, the scenario used the previous location of the robot (which was accessed for another function in the same scenario) instead of the current location; (ii) when needing to block multiple events, the scenario incorrectly issued several different YIELD commands instead of accumulating multiple blocked events in a set, and passing the set to the single yield; (iii) an IF statement that should have controlled only a few code lines, instead dominated and conditioned the rest of the scenario; (iv) a scenario that should have had a single YIELD command, waiting each time for the next move of the robot, incorrectly waited for the move both at the beginning and the end of each of its iteration. These (and other) errors were caught in testing, and the code was corrected using revised or new, corrective prompts. However, since these errors were not a result of prompt vagueness or ambiguity, one may ask how the principle of not manually modifying the code, and only changing the prompt, should be applied in this case. One approach could be to add these originally corrective actions as emphases and notes in the prompt; such notes may be reminiscent of when an experienced programmer explains a code module to a novice, sharing stories about past bugs and omissions, to emphasize the need for close attention to details.

## 5.6 Issues Around Direct Code Changes

It is often very tempting to change the code directly and not via an LLM prompt. Two examples from our experiments are changing a printed string, from “CURRENT LOCATION” to “CURRENT LOCATION:”, or inserting a PRINT() in a particular point for debug purposes. Unfortunately, such manual code changes are lost the next time the scenario is generated using the LLM, or, conversely, they cause the abandonment of the methodology.

## 5.7 Mapping Prompts to Scenarios

While in SBM scenario specifications can align with requirement specifications, our experiments show that in incremental scenario development this is not always the case. For example, initially, we separated the three functions of physically moving the robot, updating the robot's location, and printing the robot's actual path, into separate scenarios. To enforce a particular event order, we combined these three functions into one scenario (an alternative solution was to keep the scenarios separate and rely on event blocking). To generate the scenario, we prompted the LLM to create one scenario that consolidates the functions of these three scenarios in a certain order, without repeating the details of each operation. In another case, ChatGPT, on its own, added the logic that prevents the robot from stepping outside the grid boundaries, to the scenario that checks for obstacles; this created a redundancy with the scenario that was responsible for checking for grid boundaries; hence, we prompted ChatGPT to delete the unneeded scenario.

We also occasionally prompted for application-wide changes affecting all scenarios, such as changing the parameters to the YIELD command from the format that appears in the SBM preamble that we used (borrowed verbatim from [48]), to the format required by the BP Python execution environment of [47]. The entire chat dialog can then serve in having the LLM generate a concise description of each of the scenarios in the final application.

## 5.8 Emerging Requirements

One of the characteristics of agile approaches to software development is the emergence of new requirements as part of iterative sprints. In the first block of code for the robot-on-grid application, the coordinate system for grid cells that ChatGPT chose was different from how the user tacitly visualized them: (row,column) as is common in matrix description versus (horizontal,vertical), as in an (x,y) Cartesian system; in the former, a move up increases the second coordinate, and in the latter, it decreases the first coordinate. This choice, or decision, by the LLM, for an application detail that was not specified, triggered the documenting of a new requirement.

## 5.9 Low-Level Programming Choices

Some of the dialogs with ChatGPT were on specific programming style issues, like whether to rely on particular semantics of the Python language (e.g., when the compiler or interpreter exits the computation of an OR condition when one of the sub-conditions is computed to be true), or when and how to use temporary variables for streamlining the program's appearance and flow. This question is outside of the scope of the current methodology; the LLM may be taught to comply with the developer's preferences as a rule, as opposed to dealing with specific changes to already-generated code blocks.

## 6 Related Work

In recent years, LLM-based chatbots have progressed significantly, and have had a considerable impact on a diverse range of domains. Engineers and researchers are just now beginning to tap the potential applications of this technology in music [40], education [2], healthcare [38], academia and libraries [41], and many other areas.

Within the software engineering domain, which is our focus here, there have already been attempts made to harness chatbots for tasks such as evaluating the quality of code [6], correcting bugs [45], and generating code semi-automatically or completely automatically [10, 12]. The general sentiment is that chatbots will play a significant role in the future of code generation; although the specifics are not yet clear. Our work here is an attempt to outline one possible path forward, which, we believe, will help bring about this integration in a safe and controlled manner.

The methodology that we advocate here for the integration of LLMs into the software development cycle relies on the large body of existing work on the topic of scenario-based modeling [9, 33]. Concretely, we propose to leverage the SBM’s amenability to formal analysis techniques [22, 29], such as automatic repair [27], synthesis [15], and verification [20, 30]. Additional, recent work has also identified the benefits that SBM affords in this context [48]. Although we focus here on SBM, additional modeling approaches, with similar traits, could likely be used in a similar manner.

Finally, this paper can be regarded as yet another step towards the overarching vision of *Wise Computing* [23–25]. Wise computing is an attempt to transform the computer into a proactive member of the software engineering team—making suggestions and observations, raising questions, and even carrying out verification-like processes without an explicit request from a human engineer.

## 7 Conclusion

The appearance of LLMs, and the subsequent release of advanced, general-purpose chatbots, is a huge step forward, and is very likely to revolutionize the domain of software engineering in the decades to come. Still, because of errors and inaccuracies that are abundant in the outputs of such chatbots, this kind of integration must be carried out with care. Here, we outline one possible method for such an integration, which leverages the diverse capabilities of chatbots; but which at the same time also emphasizes that a careful analysis and inspection of their outputs is crucial to perform such an integration safely. It is our hope that this work will form a basis for additional research on this important topic.

As our next steps, we intend to continue this work along several directions. First and foremost, we plan to create an implementation of the environments and tools needed to fully integrate SBM and ChatGPT; and then, through this tools and environments, carry out a large, real-world evaluation of case studies

of interest. These case studies will hopefully demonstrate the usefulness of the approach as a whole.

In addition to the above, we find it highly likely that this kind of work will mandate enhancements and modifications to existing tools—chatbots and SBM frameworks alike. For example, with the current ChatGPT version, every conversation starts from a blank slate; whereas for a system that is undergoing iterative development, e.g. as part of the Wise Computing vision, it would likely be of better use to have the chatbot memorize, and utilize, previous conversations. Such an integration could be achieved, e.g., by summarizing concluded conversation and then providing these summaries back to the chatbot, as part of the preamble, whenever a new conversation is about to begin. A more elegant solution would be to customize ChatGPT specifically for the task of developing SB models and programs, and to have it remember key points of previous chats automatically. Ideally, LLMs will be able to perform such learning selectively, and over time, using their knowledge of the system being developed to determine which information from past conversation should be retained, and for how long.

Going beyond system development, the developments described above could also prove useful in a broader perspective: the same prompt engineering methods that would be useful for incremental and interactive system development could be used, e.g., to teach computers about other domains, and also in improving the training and day-to-day communications of human engineers.

**Acknowledgements.** We thank Dan Aleksandrowicz for valuable discussions and insights.

The work of Harel, Marron and Szekely was funded in part by an NSFC-ISF grant to DH, issued jointly by the National Natural Science Foundation of China (NSFC) and the Israel Science Foundation (ISF grant 3698/21). Additional support was provided by a research grant to DH from the Estate of Harry Levine, the Estate of Avraham Rothstein, Brenda Gruss, and Daniel Hirsch, the One8 Foundation, Rina Mayer, Maurice Levy, and the Estate of Bernice Bernath.

The work of Katz was partially funded by the European Union (ERC, VeriDeL, 101112713). 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 Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

## References

1. Aceto, L., Ingólfssdóttir, A., Larsen, K., Srba, J.: *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, Cambridge (2007)
2. AlAfnan, M., Dishari, S., Jovic, M., Lomidze, K.: ChatGPT as an educational tool: opportunities, challenges, and recommendations for communication, business writing, and composition courses. *J. Artif. Intell. Technol.* **3**(2), 60–68 (2023)
3. Anthropic: Claude (2024). <https://www.anthropic.com/claude>

4. Bar-Sinai, M., Weiss, G., Shmuel, R.: BPjs: an extensible, open infrastructure for behavioral programming research. In: Proceedings 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 59–60 (2018)
5. Biolchini, J., Mian, P., Natali, A., Travassos, G.: Systematic Review in Software Engineering. technical Report. System Engineering and Computer Science Department COPPE/UFRJ, Report ES 679 (2005)
6. Burak, Y., Ozsoy, I., Ayerdem, M., Tüzün, E.: Evaluating the code quality of AI-assisted code generation tools: an empirical study on GitHub copilot, Amazon CodeWhisperer, and ChatGPT (2023). <https://arxiv.org/abs/2304.10778/>
7. Chang, Y., et al.: A Survey on Evaluation of Large Language Models, technical Report (2023). <https://arxiv.org/abs/2307.03109/>
8. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of Symposium on Dynamic Languages (DLS), pp. 1–10 (2005)
9. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. *J. Formal Methods Syst. Des. (FMSD)* **19**(1), 45–80 (2001)
10. Dong, Y., Jiang, X., Jin, Z., Li, G.: Self-collaboration code generation via ChatGPT, technical Report (2023). <https://arxiv.org/abs/2304.07590/>
11. Elyasaf, A.: Context-oriented behavioral programming. *Inf. Softw. Technol.* **133**, 106504 (2021)
12. Feng, Y., Vanam, S., Cherukupally, M., Zheng, W., Qiu, M., Chen, H.: Investigating code generation performance of Chat-GPT with crowdsourcing social data. In: Proceedings of 47th IEEE Computer Software and Applications Conf. (COMP-SAC), pp. 1–10 (2023)
13. Google: Bard (2023). <https://bard.google.com/>
14. Greenyer, J., Gritzner, D., Gutjahr, T., König, F., Glade, N., Marron, A., Katz, G.: ScenarioTools—a tool suite for the scenario-based modeling and analysis of reactive systems. *J. Sci. Comput. Program. (J. SCP)* **149**, 15–27 (2017)
15. Greenyer, J., Gritzner, D., Katz, G., Marron, A.: Scenario-based modeling and synthesis for reactive systems with dynamic system structure in ScenarioTools. In: Proceedings of 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 16–23 (2016)
16. Gregorcic, B., Pendrill, A.M.: ChatGPT and the Frustrated Socrates. *Phys. Educ.* **58**(2) (2023)
17. Gritzner, D., Greenyer, J.: Synthesizing executable PLC code for robots from scenario-based GR(1) specifications. In: Seidl, M., Zschaler, S. (eds.) STAF 2017. LNCS, vol. 10748, pp. 247–262. Springer, Cham (2018). [https://doi.org/10.1007/978-3-319-74730-9\\_23](https://doi.org/10.1007/978-3-319-74730-9_23)
18. Harel, D., Assmann, U., Fournier, F., Limonad, L., Marron, A., Szekely, S.: Toward methodical discovery and handling of hidden assumptions in complex systems and models. In: Engineering Safe and Trustworthy Cyber Physical Systems—Essays Dedicated to Werner Damm on the Occasion of His 71st Birthday (2023)
19. Harel, D., Kantor, A., Katz, G.: Relaxing synchronization constraints in behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 355–372. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-45221-5\\_25](https://doi.org/10.1007/978-3-642-45221-5_25)
20. Harel, D., Kantor, A., Katz, G., Marron, A., Mizrahi, L., Weiss, G.: On composing and proving the correctness of reactive behavior. In: Proceedings of 13th International Conference on Embedded Software (EMSOFT), pp. 1–10 (2013)



21. Harel, D., Katz, G.: Scaling-up behavioral programming: steps from basic principles to application architectures. In: *Proceedings of 4th SPLASH Workshop on Programming Based on Actors, Agents and Decentralized Control (AGERE!)*, pp. 95–108 (2014)
22. Harel, D., Katz, G., Lampert, R., Marron, A., Weiss, G.: On the succinctness of idioms for concurrent programming. In: *Proceedings of 26th International Conference on Concurrency Theory (CONCUR)*, pp. 85–99 (2015)
23. Harel, D., Katz, G., Marelly, R., Marron, A.: An initial wise development environment for behavioral models. In: *Proceedings of 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 600–612 (2016)
24. Harel, D., Katz, G., Marelly, R., Marron, A.: First steps towards a wise development environment for behavioral models. *Int. J. Inf. Syst. Model. Des. (IJISMD)* **7**(3), 1–22 (2016)
25. Harel, D., Katz, G., Marelly, R., Marron, A.: Wise computing: toward endowing system development with proactive wisdom. *IEEE Comput.* **51**(2), 14–26 (2018)
26. Harel, D., Katz, G., Marron, A., Szekely, S.: On augmenting scenario-based modeling with generative AI. In: *Proceedings of 12th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 235–246 (2024)
27. Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of reactive programs. In: *Proceedings of 17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 3–12 (2012)
28. Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of safety and liveness violations in reactive programs. *Trans. Comput. Collect. Intell. (TCCI)* **16**, 1–33 (2014)
29. Harel, D., Katz, G., Marron, A., Weiss, G.: The effect of concurrent programming idioms on verification. In: *Proceedings of 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 363–369 (2015)
30. Harel, D., Lampert, R., Marron, A., Weiss, G.: Model-checking behavioral programs. In: *Proceedings of 9th ACM International Conference on Embedded Software (EMSOFT)*, pp. 279–288 (2011)
31. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play in/play-out approach. *Softw. Syst. Model. (SoSyM)* **2**, 82–107 (2003)
32. Harel, D., Marron, A., Weiss, G.: Programming coordinated scenarios in java. In: *Proceedings of 24th European Conference on Object-Oriented Programming (ECOOP)*, pp. 250–274 (2010)
33. Harel, D., Marron, A., Weiss, G.: Behavioral programming. *Commun. ACM (CACM)* **55**(7), 90–100 (2012)
34. Harel, D., Marron, A., Yerushalmi, R.: Scenario-based algorithmics: coding algorithms by automatic composition of separate concerns. *Computer* **54**(10), 95–101 (2021)
35. Harel, D., Pnueli, A.: On the development of reactive systems. *Logics Models Concurr. Syst.* **F-13**, 474–498 (1985)
36. Katz, G.: On module-based abstraction and repair of behavioral programs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) *LPAR 2013*. LNCS, vol. 8312, pp. 518–535. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-45221-5\\_35](https://doi.org/10.1007/978-3-642-45221-5_35)

37. Katz, G., Barrett, C., Harel, D.: Theory-aided model checking of concurrent transition systems. In: Proceedings of 15th International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 81–88 (2015)
38. Li, J., Dada, A., Kleesiek, J., Egger, J.: ChatGPT in healthcare: a taxonomy and systematic review, technical Report (2023). <https://www.medrxiv.org/content/10.1101/2023.03.30.23287899v1>
39. Liu, J., Xia, C., Wang, Y., Zhang, L.: Is your code generated by ChatGPT really Correct? Rigorous evaluation of large language models for code generation, technical Report (2023). <https://arxiv.org/abs/2305.01210/>
40. Lu, P., et al.: MuseCoco: generating symbolic music from text, technical Report (2023). <https://arxiv.org/abs/2306.00110/>
41. Lund, B., Wang, T.: Chatting about ChatGPT: how may AI and GPT impact academia and libraries? *Libr. Hi Tech News* **40**(3), 26–29 (2023)
42. MetaAI: LLaMa (2023). <https://ai.meta.com/llama/>
43. OpenAI: ChatGPT (2022). <https://chat.openai.com/>
44. Pettersson, O., Andersson, J.: A survey of modeling approaches for software ecosystems. In: Maglyas, A., Lamprecht, A.-L. (eds.) *Software Business. LNBIP*, vol. 240, pp. 79–93. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-40515-5\\_6](https://doi.org/10.1007/978-3-319-40515-5_6)
45. Surameery, N., Shakor, M.: Use chat GPT to solve programming bugs. *Int. J. Inf. Technol. Comput. Eng. (IJITC)* **3**(1), 17–22 (2023)
46. Vaswani, A., et al.: Attention is all you Need. In: Proceedings of 31st Conference on Advances in Neural Information Processing Systems (NeurIPS) (2017)
47. Yaacov, T.: BPPy: behavioral programming in python. *SoftwareX* **24** (2023)
48. Yaacov, T., Elyasaf, A., Weiss, G.: Boosting LLM-based software generation by aligning code with requirements. In: Proceedings of 14th International Model-Driven Requirements Engineering Workshop (MoDRE) (2024)