Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

DEVS COPILOT: TOWARDS GENERATIVE AI-ASSISTED FORMAL SIMULATION MODELLING BASED ON LARGE LANGUAGE MODELS

Tobias Carreira-Munich[1,2], Valentín Paz-Marcolla[1], and Rodrigo Castro[1,2]

[1]Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires
[2]Instituto UBA-CONICET de Ciencias de la Computación, Buenos Aires, ARGENTINA

ABSTRACT

In this paper we explore to which extent generative AI, in the form of Large Language Models such as GPT-4, can assist in obtaining a correct executable simulation model. The starting point is a high-level description of a system, expressed in natural language, which evolves through a conversational process based on user input, including suggestions for corrections. We introduce a methodology and a tool inspired by the metaphor of a copilot, a form of human-AI teaming strategy well known for its success in programming tasks. We adopt the Discrete Event System Specification (DEVS), a suitable candidate formalism that allows general-purpose simulation models to be specified in a simple yet rigorous modular and hierarchical way. The result is DEVS Copilot, an AI-based prototype that we systematically test in a case study that builds several lighting control systems of increasing complexity. In all cases, DEVS Copilot succeeds at producing correct DEVS simulations.

1 INTRODUCTION

Large Language Models (LLMs) have had great development in recent years. This generative type of Artificial Intelligence (AI) models are currently used in a vastly broad spectrum of activities such as writing essays, answering customer questions (Wang et al. 2023) or generating computer programs (Chen et al. 2021; Xu et al. 2022; Merow et al. 2023). LLM development is currently an intense area of R&D with new and sometimes unimagined applications emerging on a daily basis.

The ability of LLMs to produce quality programming code naturally raises the question of the capability of this technology to produce useful simulation algorithms, not without a plethora of concerns about their reliability, interpretability, reusability, etc. (i.e. standard criteria used in the discipline to qualify simulation systems). LLMs are also recognized as capable of producing inaccurate or incorrect answers. Additionally, the explainability/interpretability of why a given response (either correct or incorrect) was given remains a challenge (Ali et al. 2023) as LLMs constitute a black-box type of machine learning model.

In this scenario, there appears to exist a significant gap between a conceptual model, initially expressed in natural language form, and on the other end of a typical development cycle a final simulation algorithm for that model (which could be considered both accurate and explainable).

We consider that a methodologically robust attempt to bridge such a gap is to rely on sound modeling and simulation formalisms. This approach can provide a concise and unambiguous intermediate language from which to reason about and evaluate the multiple quality dimensions of the resulting simulation models.

In this work we propose the Discrete Event System Specification (DEVS) (**?**) as a suitable candidate formalism that allows general-purpose simulation models to be specified in a simple yet rigorous modular and hierarchical way. In DEVS, model structure and model behavior are clearly separated. The structure can be conceptualized as a directed graph connecting subunits that implement behavior and exchange messages with each other using the connections provided by the structure. Behavior is then defined with concise, often simple state transition functions.

Among other salient features, DEVS provides the ability to define systems that correctly combine subsystems of any type of discrete dynamics with accuracy-controlled approximations of continuous

dynamics, including both deterministic and stochastic types. Such generality and compactness combined, together with programming language agnosticism, an unambiguous abstract simulator recipe, and a minimal formal specification make DEVS a very compelling choice for exploring the limits of LLM capabilities to generate correct generalized simulation models.

There is little previous work looking at connections between natural language processing (NLP) and formal model specifications (Guan et al. 2023; Liu et al. 2022). Concerning simulation modeling Shuttleworth and Padilla (2022) explored the use of NLP to generate conceptual models from narratives. In (Giabbanelli 2023) the author describes different typical modeling and simulation tasks that LLMs could help automate, in particular on the explanation, verification, and validation of previously developed models, which are very relevant aspects that we envisage can show relevant complementarities with the methodology and tools presented in this work.

However, there is no existing methodology that helps to automate the full specification of executable simulation models departing from informal system descriptions, using LLMs as assistants (a **copilot**) to streamline the process. We aim at creating a tool that improves modelers' tasks without intervening the underlying structure of any existing DEVS simulation toolkit, relying on an LLM at its core: a copilot (Chen et al. 2021) that generates DEVS-compliant specifications, along with final simulation code, departing from natural language conceptual models. The copilot concept takes the hybrid intelligence approach, where humans and algorithms do not replace each other but are complementary instead, performing much better at the provided task than if one did the entire job alone.

The remainder of this paper is structured as follows. Section 2 delves into what LLMs, DEVS, and GPT-4 are, while Section 3 describes the proposed methodology to transform natural language descriptions of systems into a DEVS executable simulation model using the aforementioned concepts. Section 4 provides a list of usages, in increasing levels of complexity, to show the potential of the current proof of concept to understand the main features of DEVS models. Finally, Section 6 discusses the results obtained and proposes future lines of work.

2 BACKGROUND

2.1 Large-scale Pre-trained Language Models (LLMs)

Large Language Models (LLMs) are a class of Deep Neural Network (DNN) based on the *transformer architecture* concept (Vaswani et al. 2017) specifically designed for processing sequential data like natural language and characterized by their vast parameter space (typically ranging from tens of millions to over a hundred billion parameters). Transformers capture long-range dependencies and relationships within an input sequence, and unlike traditional recurrent neural networks (RNNs) or convolutional neural networks (CNNs), these rely solely on self-attention mechanisms to weigh the importance of different input tokens when processing the input sequence.

LLMs have significantly enhanced generative artificial intelligence's ability to produce code and tackle complex tasks by training on extensive datasets, encompassing code, academic papers, and the broader internet. In this work we hypothesize and systematically test the notion that the training data used for LLMs encapsulate extensive prior knowledge acquisition in the domain of simulation modeling.

Salient examples of successful LLMs include GPT-4 (OpenAI et al. 2023), LLaMA (Touvron et al. 2023) and Gemini (Gemini Team et al. 2023), to name a few. They offer a conversational interface for users, resembling chat sessions, where input *prompts* (usually expressed in natural language) are processed by the transformer. The prompts are split into an array of *tokens*, later vectorized using an *embedding*. Transformers generate a response by iteratively predicting the most probable next token (t_n). Each new token is inferred from a limited size *context* window composed by a set of all previous tokens ($t_{n-1}, \ldots, t_0$). At each new interaction, the LLM increases its conversational sessions' history via the prompts and responses that are stored in the *context*, which due to its limited size may only keep the latest tokens in this buffer, this can ultimately result in out-of-context responses.

Another important concept in LLMs is its so-called *temperature* parameter, used to reshape (usually softening) the discrete probability distribution of next tokens. This value is associated with creativity and the ability to generalize in the choice of future tokens and carries the risk of producing meaningless results. The model's outputs become more predictable as the "temperature" approaches zero, but its responses also become more conservative. The level of adequacy of such probabilistic, diverse traits of the produced responses to what is expected by a user depends largely on the type of application (ranging from arts to engineering, gaming to medicine, etc.)

Most LLMs make a strong distinction between *system prompts* and *user prompts*. The former is used to provide initial behavioral instructions, environment configurations, and general settings, while the latter represents the sequence of user interactions (acting as new inputs for the transformer).

2.1.1 The Choice of GPT-4 for LLM-assisted Model Design

OpenAI's GPT-4 stands as one of the most ubiquitous and capable LLMs currently available. Several others exist, each with different versions, capabilities, and licensing schemes.

In this work, as we shall see, a key requirement for adopting an LLM is its ability to produce quality code in the Python language (Hou and Ji 2024) and to offer the largest possible *context* token size (we envision this methodology to be used on much larger examples than those experimented with in this work, including more elaborate system model descriptions).

GPT-4 offers these features and, in addition, provides a robust and massively tested API interface. At the time this decision was made Claude 3 was not yet released, Claude 2 was not efficient enough, Google's Gemini Pro had a maximum of 30k token *context*, LLaMA 2 4k and CodeLLaMA 16k. We will not be using any features unique to GPT-4 for our experiments, so replacing GPT-4 (specifically `gpt-4-0125-preview` with a 128k token *context*) with a similar alternative should be transparent. As for fine-tuned LLM models: they usually perform much better than their standard counterparts. However, this option has not been used in this work to better compare results against other LLMs in the future.

LLMs provide different hyperparameters to control randomness and internal sampling. For this paper, the *temperature* was fixed and set to 0.5, to give GPT-4 creativity without compromising much on coherence (**?**). All experimental results should be interpreted as specific to this hyperparameter value, while a detailed parameter sweeping remains a subject of future work.

2.2 Formal Simulation Modeling with DEVS

The DEVS formalism (**?**) allows expressing simulation models that are based on discrete events. DEVS defines two main constructs: *atomic* and *coupled* models. The elements of these constructs shall be reflected in a BNF grammar for classic DEVS proposed in Section 3.3.

An atomic model is defined by the tuple $M =< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta >$, where X is the set of input event values, Y is the set of output event values, and S is the set of state values. $\delta_{int} : S \to S$ is the internal transition function, $\delta_{ext} : S \times \Re_{\geq 0} \times X \to S$ is the external transition function, $\lambda : S \to Y$ is the output function, and $ta : S \to \Re_{\geq 0}$ is the time advance function.

DEVS models can be coupled together in modular and/or hierarchical ways. A DEVS coupled model (CN) is defined by the following tuple structure: $CN =< X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, Select >$, where X and Y are the sets of input and output event values, D is the set of component (atomic or coupled) references. For each $d \in D$, M_d is a DEVS model and I_d is the set of influences of model d. *Select* function is the tie-breaking selector. As $d \in D$ components can also be of type coupled, hierarchical (multilevel) constructs can readily be defined. A unique *root* coupled model serves as a default *universe* (a top-level container) for the entire system.

3 PROPOSAL OF THE AI-ASSISTED DEVS MODELING METHODOLOGY

In this section we describe our proposed methodology to generate DEVS simulation models from natural language descriptions of a system. We split the pipeline into two subsequent stages: the `Copilot` stage (3.1), where the user interacts conversationally with the LLM to produce `PythonPDEVS` simulation models and their corresponding system specifications; and the simulation stage where the concrete simulator is run producing simulation results. The overall user flow is as follows (see Figure 1):

1. The modeler creates a **system description** explaining the system, its components, and their relationships. This need not be DEVS oriented, but a natural language conceptual system explanation.
2. The `Copilot` application is run, presenting to the modeler the intermediate **specifications** produced and the candidate `PythonPDEVS` simulation model generated.
3. If the modeler finds any of the answers unsatisfactory, they should interact again with the `Copilot` providing new prompts in natural language requesting for the intended corrections (back to step 2)
4. The modeler is satisfied with the produced specifications. Run the produced simulation model.
5. Examine the simulation results. If unsatisfactory, give the `Copilot` new instructions, corrections or requirements. In such a case, go back to step 2. Otherwise, the process ends satisfactorily.

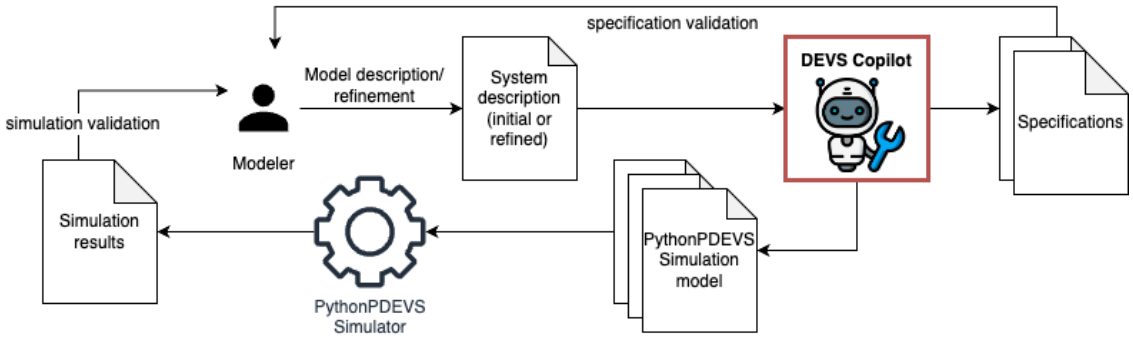


Figure 1: Proposed AI-assisted interactive and iterative modeling and simulation process.

`PythonPDEVS` (Van Tendeloo and Vangheluwe 2014) is our concrete simulator of choice, and as a first approach, we will focus on Classical DEVS definitions only. As there is currently no standardized DEVS specification language (Blas et al. 2023; Hong and Kim 2006; Kim and Kim 1999) we propose a custom DEVS-compliant language for the intermediate specifications (see details in 3.3).

3.1 The DEVS-Copilot Component

The DEVS `Copilot` is an application that issues 2 different calls to the OpenAI's GPT-4 API. This sequencing is aimed at splitting the *semantic understanding* from the *syntactic code generation*, allowing for more targeted corrections by the modeler, thus providing a controlled, multi-stage LLM-assisted process. For example: a field ordering problem in a specification is not conceptually equivalent to the model behaving incorrectly, and should not be treated in the same way.

The first component is the **Concept Specifier**, intended to transform a user-provided *system description* (its *user prompt*) and enrich it with DEVS meta-model information, remove redundant information, and disambiguate model relations and internals, finally generating a semi-structured *DEVS conceptual specification* (**conceptual spec** for short). This result should be human-readable, possibly having some ambiguity, but much closer to a formal, correct, and complete description of the system, giving an idea of the LLM's ability to understand what a DEVS conceptual model that fits the described system looks like.

The second component is the **Formal Specifier**, intended to produce a *formal DEVS specification* (**formal spec** for short) which is a proposed structured and parsable formal specification, a form of Domain Specific Language (DSL). Its *user prompt* is the **conceptual spec** produced at the previous stage,

while its **system prompt** provides a grammar and other specifics about `Python` (the base language of `PythonPDEVS`). Thus, this second stage is independent of the prior, and focuses only on the target simulator and specification.

If we changed the target simulator language or the specification grammar, only the **formal spec** would need adjustments, making the `Copilot` more general than if both stages handled intertwined concerns. For example, it is reasonable to assume that if a given **conceptual spec** produces a correct specification, it would also produce a correct result for some other target simulator or specification language. An example of a **conceptual spec** can be seen in Listing 1, that corresponds to the **formal spec** in Listing 2, both part of the System 1 presented next in the results Section 4. The source code of the `Copilot` , along with the results of this work, can be found in https://git-modsimu.exp.dc.uba.ar/valchulen/DEVS_Copilot.

Listing 1: System 1 - Conceptual specification.

```
### Root Model
- The root model is named "root".
- It contains two atomic models: "Controller" and "Light".
- The "Controller" sends signals to the "Light".
- Connections:
- From "Controller" output port "signal" to "Light" input port "command".
### Controller Atomic Model
- **Ports**:
- Output: "signal" (sends "on" or "off" commands).
- **State Variables**:
- `currentState`: Indicates whether the next signal to send is "on" or "off". Initial value is "on".
- `frequency`: Determines the rate at which signals are sent. Initial value is 1Hz.
- `timeAdvance`: Time advance variable, initially set to 10 seconds to start the cycle.
- **Internal Transition Function**:
- Toggles `currentState` between "on" and "off".
- Resets `timeAdvance` to 1/frequency seconds after each toggle.
- **External Transition Function**:
- Not applicable as the controller does not respond to external inputs in this scenario.
- **Output Function**:
- Sends the value of `currentState` ("on" or "off") through the "signal" port.
### Light Atomic Model
- **Ports**:
- Input: "command" (receives "on" or "off" commands).
- **State Variables**:
- `status`: Indicates the current state of the light ("on" or "off"). Initial value is "off".
- `timeAdvance`: Set to infinity since the Light model does not have self-triggered actions.
- **Internal Transition Function**:
- Not applicable as the light's state changes are only triggered by external inputs.
- **External Transition Function**:
- On receiving a "command":
  - Updates `status` to reflect the received command ("on" or "off").
- **Output Function**:
- Not applicable as the Light model does not generate outputs.
### Model Instances and Connections
- Instances within the "root" coupled model:
- One instance of the Controller atomic model, named "ControllerInstance".
- One instance of the Light atomic model, named "LightInstance".
- Connections:
- From "ControllerInstance" output port "signal" to "LightInstance" input port "command".

This setup describes a discrete event simulation where a controller toggles a light on and off at a frequency of 1Hz, starting the cycle 10 seconds after the simulation begins, and continuing indefinitely.
```

3.2 Copilot variants: adding memory to the process.

It is widely known that LLMs are quite good at generating programming code (Chen et al. 2021; Xu et al. 2022; Merow et al. 2023), so the focus of this work is not on deep-diving into the ways in which LLMs generate a **formal specification**, but rather on grasping their formal modeling capabilities. In that spirit, the `Copilot` has two variants that differ only in the *conversation history* content of **Formal Specifier**.

Listing 2: System 1 - Formal specification.

```
# Controller Atomic Model
[Controller]
in_ports = []
out_ports = ["signal"]
time_advance = 10
internal_transition_function = """
if currentState == "on":
    currentState = "off"
else:
    currentState = "on"
timeAdvance = 1 / frequency
"""
output_function = """
send(signal, currentState)
"""

[Controller.state.currentState]
type = "string"
initial_value = "on"

[Controller.state.frequency]
type = "float"
initial_value = 1.0

# Light Atomic Model
[Light]
in_ports = ["command"]
out_ports = []
time_advance = inf
external_transition_function = """
if port_name == "command":
    status = message
"""
[Light.state.status]
type = "string"
initial_value = "off"

# Root Coupled Model
[root]
[[root.component]]
model = "Controller"
name = "ControllerInstance"
[[root.component]]
model = "Light"
name = "LightInstance"
[[root.connections]]
source = "ControllerInstance.signal"
target = "LightInstance.command"
```

The simplest variant (see Figure 2), identified as **without formal specifier history** (or **without history** for short), treats every interaction with the **Formal Specifier** as isolated from all prior prompts and conversations. This is, each call to this LLM considers only the current **user prompt**, the fixed **system prompt** (that includes the instructions on what it should generate), and the formal grammar (with Python as the base language). This limits this stage to syntactical understanding only, using the conceptual model structure provided as input. We will show that this variant makes the same mistakes even when given a correction in a previous interaction.

The second variant (see the **blue** cloud in Figure 2), named **with formal specifier history** (or **with history** for short), provides the **Formal Specifier** with the entire conversational history of **formal specs** it has produced. We will show that this variant simplifies and reduces the number of refinement interactions when trying to refine specifications.

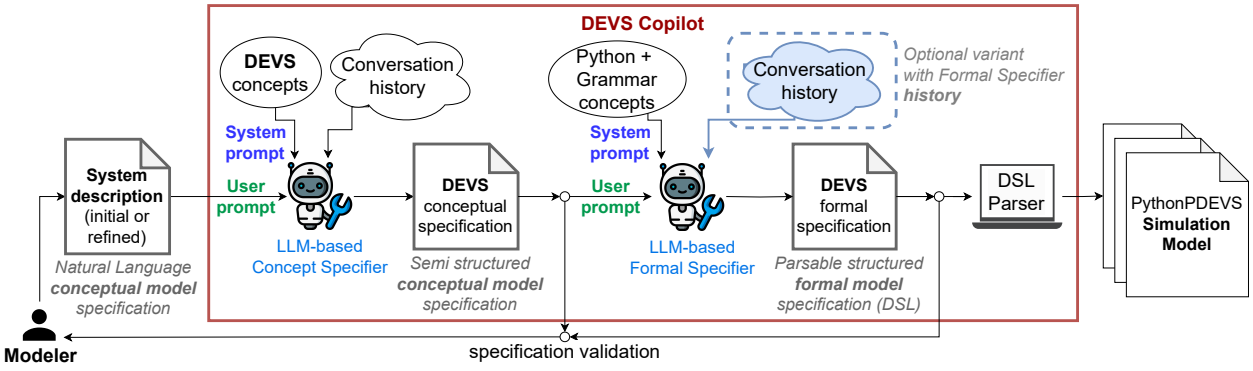


Figure 2: Copilot pipeline & variants.

3.3 Specification

Our goal is to explore and describe the capabilities of LLMs to understand and use core DEVS concepts like action-reaction, modularity, extensibility, hierarchies, and time management. Note that the grammar of **formal spec** (in BNF form, available in https://git-modsimu.exp.dc.uba.ar/valchulen/devs_copilot/-/blob/main/ClassicDEVS_BNF.grammar) does not depend on the `PythonPDEVS` specific simulator but on Python, the base language for this toolkit, because its data types and the code within DEVS functions used for simulation must match that of the toolkit. If another target simulation toolkit were to be chosen, these code fragments should be changed to match the appropriate new language. Therefore, GPT-4 should only generate Python-specific code (for the DEVS formal specification), a process that has been extensively tested by the coding community worldwide.

Previously suggested DEVS specifications exist (Blas et al. 2023; Hong and Kim 2006; Kim and Kim 1999) covering various aspects of the modeling process. However, we find them overly complex for the purposes of this work. The DEVS specification was made intentionally minimal, to streamline potential corrections and steer the LLM's focus to each problem at hand.

The grammar describes the specification language used to generate, with a parser program, all the required code for simulations, and is part of the **formal specifier**'s *system prompt*. Here are a few notable aspects of the language:

- It is restricted to TOML for ease of reading, parsing, and LLM understanding. We don't need to explain the TOML rules, which are assumed as already known by GPT-4.
- The functions body are specified as arbitrary Python code. State variables must be referenced by name. State variable types are currently restricted to string, float, and integer, but could be extended.
- Instead of an atomic model having a time advance function, it has a time advance state variable (referenced as `ta`) with initial value (`time_advance`). It changes via simple variable assignment whenever a state change should impact the time advance value, and is read by the time advance function. This is a common practice in DEVS modeling.
- The requirements also mandate the inclusion of a "root" model to ensure a parser can establish a correct entry point for generating a simulation.

4 CASE STUDY: SWITCHING LIGHTS ON AND OFF TO TEST DEVS CORE FEATURES

We propose a list of example systems under study, which will serve as an overall proof of concept for the proposed workflow (see Figure 3). These seek to test the main features of DEVS as a modeling and simulation formalism. We start with a classic and simple case where a light is turned on and off controlled by an event generator. Then, we make this seed example increasingly complex to exercise modularity, hierarchy and port mapping, among other DEVS features.

Each case study (except for the first one) is built as an evolution of a previous example chosen by its suitability as a convenient starting point for the new system, following an iterative-incremental process expected both for LLM conversations and for simulation model building. Technically, the knowledge about any previous **Base System** resides in the **conversation history** used at each LLM-based specifier.

We propose the following criteria of acceptability to categorize the outcomes of the LLM-based process:

- **Acceptable**: The specification is accurately parsed into a simulation that executes as expected, requiring no further intervention from the user.
- **Recoverable**: While the underlying logic is sound, there is at least one syntax error (unparsable) that needs manual correction by the user (for example the issue described in Section 5.2).
- **Incorrect**: This result indicates a logical error, which could stem from various sources, such as the meta-model (e.g., incorrectly expecting the `output_function` to be invoked after the `external_transition_function`, see Section 5.4), the specification itself (e.g., incorporating non-Python code within functions), or the system description (e.g., failing to deactivate a light when required, see Section 5.1). These errors point to fundamental misconceptions within the

`Copilot`'s operation and require manual user intervention (rather than LLM-driven regeneration). **Formal specs** of this type may be unparsable.

After the `Copilot` generates a specification we look for errors in the results (currently a direct visual inspection, with no extra checkers involved). If the result was **incorrect** we interact with the tool again, explaining in natural language the problem found and suggesting how to fix it. Each of these interactions is called a **refinement**. If we find no issues in a result (i.e. the specification is **acceptable**) it is parsed into a simulation code ready to run. Otherwise, every other error in the specification is **recoverable**, so no more **refinements** are necessary, but someone has to manually go and fix said issues before the specification is **acceptable** and therefore can run.

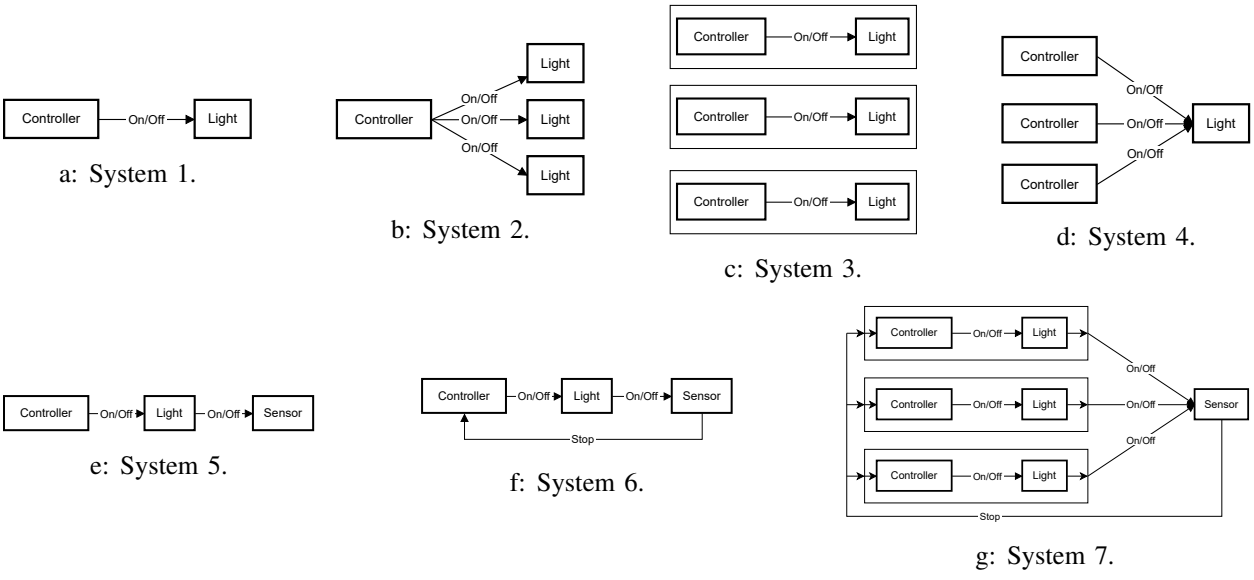
5 EXPERIMENTAL RESULTS: ANALYSIS OF SYSTEMS, PROMPTS AND REPRESENTATIVE INTERACTIONS

Table 1 summarises the main features of the experimentation process using DEVS Copilot. Each new System ID is created from a given System Description User Prompt, uses some previous Base System as a starting point, employs variants of the Formal Specifier with/without history, and resorts to a certain number of refinement cycles. **For all systems, the final executed simulations produce the correct expected results** (in the form of output logs, not shown here due to space limitation and the simplicity of the systems).

Next, we showcase snippets of results that we found representative of *typical interactions* with the `Copilot`. Each shows a particular type of error found and, when applicable, which extra *user prompt* was used to fix it. As a starting point, refer to Listing 2 to see the specification generated for System 1.

5.1 Example Copilot Interaction #1: Semantic Error

In System 3 (with history, 2^{nd} interaction): After clearly stating that *frequency* is a parameter that should be set to a random value at the start of the simulation and that it shouldn't change, the `Controller.internal_transition_function` resets it (see Listing 3). With the new *user prompt* detailed in Listing 4 used as subsequent interaction, the issue is fixed and doesn't show up again, with `frequency=rand()` being removed by the Formal Specifier from the internal transition function.



a: System 1. b: System 2. c: System 3. d: System 4. e: System 5. f: System 6. g: System 7.

Figure 3: Proposed systems under study.

Table 1: Analysed prompts and systems.

System ID	System Description User Prompt	Base System ID	Variant	Refinement cycles
1	We have a module named controller that sends on and off signals to a light. The controller must send these commands with a parametric frequency with default initial value of 1Hz. The initial state of the light is off and the controller will auto-start the cycle 10 seconds after the simulation start. Once started the toggling should never end.	N/A	N/A	0
2	Now extend the system so a single controller toggles 5 lights simultaneously. †	1	wo/history	0
			w/history	0
3	Now create a controller-light coupled model that can be used as a single unit. Of this controller-light abstraction, create 5 pairs. Modify the base controller to have a random frequency (you can use the "rand()' python function). †	1	wo/history	1
			w/history	0
4	Now extend the system so that there are 5 controllers sending signals to a single light. Change the controller model such that the parameterized frequency is random. †	1	wo/history	1
			w/history	1
5	Now add a sensor atomic model able to know if the light is on or off. †	1	wo/history	1
			w/history	2
6	Now please modify the models and their relationships so that the sensor can send a signal to the controller; telling it to stop toggling the light and simply turn it off. This should happen after 10 on-off light cycles, the sensor is the component responsible of knowing when that threshold is reached. †	5 w/history	wo/history	1
			w/history	1
7	Now change the system such that: - "Controller" and "Light" are coupled in a new model, called "ControllerLight" - There are 3 different "ControllerLights" in the system - The controller model has a random frequency parameter (you can use the "rand" function) - There is only one "Sensor", it should sense all lights - After the "Sensor" has seen 10 total light toggles (does not matter from which light the signal arrived) it sends a stop signal to every controller. - A full light cycle happens when lights go from "on" to "off" states. - After the stop signal is sent, lights should be turned off. †	6 w/history	wo/history	3
			w/history	1

A † represents the addition of the following text: "Do not forget to also specify the internal functions, state variables, and other variables for all atomic models in the system, even if you have to repeat information."

Listing 3: Ex. #1 - Formal specification snippet.

```
...
internal_transition_function = """
...
frequency = rand()
ta = 1 / frequency
"""
...
```

Listing 4: Ex. #1 - User prompt.

```
Great response, but the frequency must
be set to a random value when the
simulation begins; then it should stay
fixed.
```

5.2 Example Copilot Interaction #2: Syntactic Error

In System 6 (5^{th} interaction), System 4 (3^{rd} interaction), and System 2 (2^{nd} interaction), all without history: a very frequent error (particularly in **without history** variants) is that `time_advance`, `output_function`, `internal_transition_function` and/or `external_transition_function` are not located in an atomic model's main scope, but after its state variable scope (see Listing 5). This contradicts the grammar and wouldn't parse, but represents a

simple syntactical error. We consider responses with this type of error as **recoverable**, so no extra *user prompt* was required. Also, a similar error can be found in System 3 (without history, 3^{rd} interaction). Of course, this error type is undesirable and we favor variants where this is less prone to happening.

Listing 5: Ex. #2 - Formal specification snippet.

```
...
["Light"]
in_ports = ["signal_in"]

[Light.state.state]
type = "string"
initial_value = "off"
external_transition_function = """
if port_name == "signal_in":
    if message == "on":
        state = "on"
    elif message == "off":
        state = "off"
"""
...
```

5.3 Example Copilot Interaction #3: Meta-model Semantic Error 1

In System 6 (with and without history, 5^{th} interaction): the `Sensor` must notify the `Controller` that it has identified 10 total on-off cycles, thus in the **formal spec** when the threshold is reached, `output_function` is triggered by setting *ta* to 0. But as there is no `internal_transition_function`, *ta* is not reset to a non-instantaneous value so the simulation loops infinitely. See line 9 in Listing 6.

We consider this a misconception of the understanding of the DEVS formalism, but with a *user prompt* describing the issue and the "correct it" instruction (Listing 7), an `internal_transition_function` is added to the **formal spec** setting $ta = float('inf')$ when the cycling has finished.

Listing 6: Ex. #3 - Formal specification snippet.

```
...
[Sensor]
external_transition_function = """
if port_name == 'status':
    lightStatus = message
    if lightStatus == 'off':
        cycleCount += 1
    if cycleCount >= 10:
        ta = 0
    else:
        ta = float('inf')
"""
...
```

Listing 7: Ex. #3 - User prompt.

```
The sensor loops infinitely when
the cycle count is reached.
Correct the 'ta'.
```

5.4 Example Copilot Interaction #4: Meta-model Semantic Error 2

In System 7 (without history, 8^{th} interaction): here the opposite of what is shown in Example interaction #3 happens, `Sensor.output_function` is never triggered, because the `Sensor` never has $ta < +\infty$. See `Sensor.external_transition_function` in Listing 8.

This is a conceptual error stemming from a misconception in the understanding of the meta-model, that is fixed by the *user prompt* from Listing 9: the conditional $ta = 0$ is added to the function, and the infinite looping is avoided by resetting the $ta = \infty$ in `Sensor.internal_transition_function`.

Listing 8: Ex. #4 - Formal specification snippet.

```
...
["Sensor"]
external_transition_function = """
if port_name == "status" and message == "off":
    toggleCount += 1
"""
output_function = """
if toggleCount == 10:
    send("control", "stop")
"""
...
```

Listing 9: Ex. #4 - User prompt.

```
output_functions are not run after
 external_transition_functions,
use the ta = 0 trick accordingly.
Make sure you are not introducing
infinite loops when using this
technique.
```

6 DISCUSSION, CONCLUSIONS AND OUTLOOK

In this paper we have systematically explored the potential of using LLMs to generate formal and correct DEVS simulation models. The results are very promising and suggest that the Copilot metaphor, together with the proposed methodology, can play an important role in the modeling discipline.

All user prompts utilized for model refinement and error correction were written in a general conversational tone, addressing the conceptual problem at hand but without specifying a technical solution. We also deliberately used prompts with a low/medium level of complexity, compatible with what an undergraduate student with basic training in systems modeling and simulation would produce, and refrained from using any advanced type of prompt engineering technique (Xu et al. 2022). These points, combined with the proven success of DEVS Copilot in correcting issues (if any) in just a few iterations, support the argument that our LLM-based specifiers accurately interpret the underlying problems. In future studies we will test the system using a collection of prompts proposed by undergrad students before and after taking a DEVS simulation course, to draw more insightful conclusions.

Although we deliberately adopted Classic DEVS to keep this first approach as manageable as possible, it is of interest to broaden the scope using DEVS extensions. Natural candidates are Parallel DEVS (**?**) and Emergent Behaviour DEVS (Foguelman et al. 2021), both because of their relevancy to research and because they are already available in `PythonPDEVS`. As DEVS Copilot is simulator-agnostic, it is clear that experiments with other toolkits written in different languages should be conducted.

Through experimentation, we were able to verify a well-known fact: LLMs are currently not reliable enough to be part of a fully automated tool if correctness and reliability are a must. They are often not able to respect the requested formats, as we have seen in the examples. For instance, when a modification of a previous response is requested, the unmodified atomic models are not detailed and instead the response lists only the differences relative to the previous model. While this would be convenient when communicating with a human, it makes it difficult to algorithmically extract the full specification from the LLM responses in a deterministic way.

However, it should be possible to safely automate certain processes in the proposed scheme. The correction of models that do not conform to the provided grammar is particularly important. By implementing a parser that generates clear error messages, DEVS Copilot could automatically iterate a specification until it is generated correctly. Another area that requires research is the automatic verification of simulation results based on LTS/LTL specifications. These properties may be generated by the same or a different LLM specifier in the pipeline.

On a more speculative note, we can reflect on the role of simulation models in the field of Artificial General Intelligence (AGI) (Goertzel 2014). Current AI exhibits narrow or specialised intelligence, capable of excelling at specific tasks but lacking broad understanding and reasoning. Formal models in general, and formal simulation models in particular, serve as a means of reasoning about reality in an unambiguous and structured way. Thus, instead of asking AI to achieve generality, an alternative way might be to guide AI to think in terms of general-purpose modelling formalisms that humans already know how to interpret and explain. Of course, the stress would then be put on the modelling formalism of choice. This is where DEVS can show its full potential, by demonstrating its ability to correctly represent models of a wide variety, and to express complex hybrid system models built by combining sub-models of heterogeneous types, thus promoting the desired generality. As we move forward, the pathway of the integration of AI with formal simulation modelling will unfold over time, revealing its full potential impact and limitations.

REFERENCES

Ali, S., T. Abuhmed, S. El-Sappagh, K. Muhammad, J. M. Alonso-Moral, R. Confalonieri *et al*. 2023. "Explainable Artificial Intelligence (XAI): What we know and what is left to attain Trustworthy Artificial Intelligence". *Information Fusion* 99:101805.

Blas, M. J., S. Gonnet, D. Kim, and B. P. Zeigler. 2023. "A Context-Free Grammar for Generating Full Classic DEVS Models". In *2023 Winter Simulation Conference (WSC)*, 2579–2590 https://doi.org/10.1109/WSC60868.2023.10407991.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan *et al*. 2021. "Evaluating Large Language Models Trained on Code". *arXiv preprint arXiv:2107.03374*.

Foguelman, D., P. Henning, A. Uhrmacher, and R. Castro. 2021. "EB-DEVS: A formal framework for modeling and simulation of emergent behavior in dynamic complex systems". *Journal of Computational Science* 53:101387.

Gemini Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut *et al*. 2023. "Gemini: A Family of Highly Capable Multimodal Models". *arXiv preprint arXiv:2312.11805*.

Giabbanelli, P. J. 2023. "GPT-Based Models Meet Simulation: How to Efficiently use Large-Scale Pre-Trained Language Models Across Simulation Tasks". In *2023 Winter Simulation Conference (WSC)*, 2920–2931 https://doi.org/10.1109/WSC60868.2023.10408017.

Goertzel, B. 2014. "Artificial General Intelligence: Concept, State of the Art, and Future Prospects". *Journal of Artificial General Intelligence* 5(1):1–48.

Guan, L., K. Valmeekam, S. Sreedharan, and S. Kambhampati. 2023. "Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning". In *Advances in Neural Information Processing Systems*, Volume 36, 79081–79094. Red Hook, United States: Curran Associates Inc.

Hong, K. J. and T. G. Kim. 2006. "DEVSpecL: DEVS specification language for modeling, simulation and analysis of discrete event systems". *Information and Software Technology* 48(4):221–234.

Hou, W. and Z. Ji. 2024. "A systematic evaluation of large language models for generating programming code". *arXiv preprint arXiv:2403.00894*.

Kim, H. and J. Kim. 1999. "An XML-based DEVS Markup Language for Sharing Simulation Models on the Web". *Journal of the Korea Society for Simulation* 8(1):113–138.

Liu, J. X., Z. Yang, B. Schornstein, S. Liang, I. Idrees, S. Tellex *et al*. 2022. "Lang2LTL: Translating Natural Language Commands to Temporal Specification with Large Language Models". In *Workshop on Language and Robotics at CoRL 2022*. December 14th-18th, Auckland, New Zealand.

Merow, C., J. M. Serra-Diaz, B. J. Enquist, and A. M. Wilson. 2023. "AI chatbots can boost scientific coding". *Nature Ecology & Evolution* 7(7):960–962.

OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya *et al*. 2023. "GPT-4 Technical Report". *arXiv preprint arXiv:2303.08774*.

Shuttleworth, D. and J. Padilla. 2022. "From Narratives to Conceptual Models via Natural Language Processing". In *2022 Winter Simulation Conference (WSC)*, 2222–2233 https://doi.org/10.1109/WSC57314.2022.10015274.

Touvron, H., T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix *et al*. 2023. "LLaMA: Open and Efficient Foundation Language Models". *arXiv preprint arXiv:2302.13971*.

Van Tendeloo, Y. and H. Vangheluwe. 2014. "The modular architecture of the python(P)DEVS simulation kernel: work in progress paper". In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*, DEVS '14. San Diego, United States: Society for Computer Simulation International.

Vaswani, A., Google Brain, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones *et al*. 2017. "Attention Is All You Need". In *Advances in Neural Information Processing Systems*, 6000–6010. Red Hook, United States: Curran Associates Inc.

Wang, L., C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang *et al*. 2023, 8. "A Survey on Large Language Model based Autonomous Agents". *Frontiers of Computer Science* 18(6):1–26.

Xu, F. F., U. Alon, G. Neubig, V. J. Hellendoorn and V. J. Hel. 2022. "A systematic evaluation of large language models of code". In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10. New York, United States: Association for Computing Machinery (ACM).

AUTHOR BIOGRAPHIES

TOBIAS CARREIRA-MUNICH is a PhD candidate in Instituto de Ciencias de la Computación at Universidad de Buenos Aires. His email address is tcarreira@dc.uba.ar.

VALENTÍN PAZ-MARCOLLA is an advanced bachelor's thesis student in Computer Science at the Departamento de Computación, Universidad de Buenos Aires. His email address is valentinpazm@gmail.com.

RODRIGO CASTRO is a Professor in the Departamento de Computación, Universidad de Buenos Aires, and Head of the Laboratory on Discrete Event Simulation at the CONICET Research Institute of Computer Science (ICC). His research interests include simulation and control of hybrid systems. His email address is rcastro@dc.uba.ar.