Proceedings of the 2025 Winter Simulation Conference
E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.

GENERATIVE STATECHARTS-DRIVEN PDEVS MODELING

Vamsi Krishna Vasa[1], Hessam S. Sarjoughian[1], Edward J. Yellig[2]

[1]Arizona Center for Integrative Modeling & Simulation, Arizona State University, Tempe, AZ, USA
[2]Intel Corporation, Chandler, AZ, USA

ABSTRACT

Behavioral models of component-based dynamical systems are integral to building useful simulations. Toward this goal, approaches enabled by Large Language Models (LLMs) have been proposed and developed to generate grammar-based models for Discrete Event System Specification (DEVS). This paper introduces PDEVS-LLM, an agentic framework to assist in developing Parallel DEVS (PDEVS) models. It proposes using LLMs with statecharts to generate behaviors for parallel atomic models. Enabled with PDEVS concepts, plausible facts from the whole description of a system are extracted. The PDEVS-LLM is equipped with grammars for the PDEVS statecharts and hierarchical coupled model. LLM agents assist modelers in (re-)generating atomic models with conversation histories. Examples are developed to demonstrate the capabilities and limitations of LLMs for generative PDEVS models.

1 INTRODUCTION

A system's description is the first thing needed to begin creating a simulatable model. The system description should provide some notional understanding of the system and its expected simulated behavior. Modelers also require knowledge of the concepts and methods of a modeling language to identify and specify the structural and behavioral aspects of a system. In this context, Large Language Models (LLMs) equipped with suitable capability and knowledge can assist in developing simulation models. Natural Language Processing (NLP) is proposed to generate formal models from textual description (Liu et al. 2022). Indeed, Machine Learning, similar to many other scientific research, is expected to support the modeling, simulation, and experimentation lifecycle activities. However, generating correct and accurate models using machine learning is challenging. Such models should be created based on sound modeling principles, grounded in mathematical formalisms, and implementable in simulators.

Given the numerous roles machine learning plays in modeling and simulation, this paper focuses on transformer-based frameworks that can generate state-based operations from system descriptions. Generated models are expected to be correct and faithfully simulate time-indexed behaviors. An LLM framework is proposed for generating executable models from system descriptions (Jackson et al. 2024). As an GPT-3 Codex framework, it assists in creating and simulating a discrete-time model of an inventory from a prompt describing what it is and how it works. The GPT-4 (Wang et al. 2023), supported with a grammar for the classic Discrete Event System Specification (DEVS) modeling formalism (Zeigler, Muzy, and Kofman 2018), is also proposed (Carreira-Munich et al. 2024). This research shows the importance of using formal modeling and Co-Pilot to develop an LLM-based framework for generating PythonPDEVS simulation code (L. 2018). Similarly, the GPT-4 API is used to assist with generating DEVS-based activity models (Alshareef et al. 2024). They show the use of transformer-based LLMs capable of generating models with varying capabilities and limitations.

Taking into account the above observation, this paper proposes using Parallel DEVS (PDEVS) (Zeigler et al. 2018) and PDEVS statecharts (Fard and Sarjoughian 2015) to develop an LLM-based framework to generate models from textual descriptions. Grammars based on the PDEVS statecharts and atomic model specification are defined to model the behaviors of atomic models given system textual descriptions.

Similarly, a grammar for the PDEVS hierarchical coupled models is defined. The resulting LLM framework supports statecharts-based atomic models and hierarchical coupled models. The contributions of this paper can be summarized as follows: *(i)* an agentic LLM framework for PDEVS statecharts with conversation histories, *(ii)* hierarchical PDEVS model structures, and *(iii)* curated sample hierarchical system descriptions with their generated PDEVS models demonstrating some of their capabilities and limitations.

2 BACKGROUND

2.1 LLMs as AI Agents

Recent advancements in the generative capabilities of transformer-based LLMs have enabled their application to various intellectual tasks (Fan et al. 2024). They are demonstrated to be capable of performing tasks such as code generation from textual descriptions. These models can be deployed in a chat-based (*user prompts*) environment, incorporating a memory element that enables users to have continuous conversations. The prompts are first divided into an array of tokens, which are then vectorized using embeddings. The LLMs then generate responses by iteratively predicting the most likely next token (k_n) given a set of preceding tokens ($k_{n-1}, \cdots, k_0$). Each new token is inferred based on a finite size context window. These models have led to a new branch of research, where researchers instruct LLMs by providing *system prompts*. As LLM agents, they are inherently able to adapt their behavior and response strategies based on the roles and constraints encoded in these prompts, enabling context-sensitive reasoning, task alignment, and persona-driven interactions across diverse domains (Gao et al. 2024; Liu et al. 2023).

2.2 Modeling with Parallel DEVS

Discrete and event-based systems can be modeled using the Parallel DEVS modeling formalism. Its modular, hierarchical foundation is defined as two complementary abstract algebraic structures, one for atomic models and the other for coupled models. Atomic models represent the fundamental building blocks of a system. Coupled models are composed of multiple atomic models (or other coupled models) interconnected to define the overall system. Atomic models are specified as $\langle X_M^b,\ Y_M^b,\ S,\ \delta_{ext},\ \delta_{int},\ \delta_{con},\ \lambda,\ ta\rangle$ where X_M^b and Y_M^b are independent sets of input and output ports with arbitrary values, S is a set of states, δ_{ext} is the external state transition function, δ_{int} is the internal state transition function, δ_{con} is the confluence transition function, λ is the output function, and ta is the time advance function. The specification for the coupled model is available (Zeigler, Muzy, and Kofman 2018).

The abstract atomic model can be made concrete using PDEVS statecharts which is based on UML statecharts. The δ_{ext} and δ_{int} are defined as EXTERNAL-EVENT[GUARD-CONDITION]$_{OPT}$ / ACTIONS and INTERNAL-EVENT[GUARD-CONDITION]$_{OPT}$ / ACTIONS for pairs of source and target states, respectively. The statechart state is used to define λ function as OUTPUT(PORT, MESSAGES)[GUARD-CONDITION]$_{OPT}$ / ACTIONS. The output functions can be assigned to state transitions entering a state or state self-transitions. The PDEVS external and output functions have input and output ports, respectively. The statechart specification includes an initialization function for every atomic model's initial state $S_i \subseteq S$. Select elements for the statechart of a basic processor are provided below:

S: PHASE = {ACTIVE, IDLE, START}
δ_{ext}: !(PORT-IN, JOB)[PHASE == IDLE] / PHASE = ACTIVE, @ PTIME
δ_{int}: [PHASE == ACTIVE] / PHASE = IDLE, @ INFINITY
λ: ?[PHASE == ACTIVE] / CONTAINER.ADD(JOB)

The START, IDLE, and ACTIVE are the possible state values for the state variable PHASE. The START serves as the initial state. The ! is used for incoming events. The ? is used for outgoing events. The ADD action adds the event JOB to a container. Containers are necessary to produce multiple output events on multiple output ports at the same time. The @ signifies the amount of logical time allocated to external and internal event state transition functions. In this example, PTIME is the processing time.

2.3 Related Work

Machine learning in the context of supporting simulation modeling has been of interest from various standpoints. The use of NLP for generating conceptual models from textual description is studied (Shuttleworth and Padilla 2022). It describes identifying and extracting model elements from text. From a broader perspective, LLMs can simplify and streamline common activities (e.g., comparing/analyzing simulation outputs) in the modeling and simulation lifecycle (Giabbanelli 2023).

In another work, it is proposed to use reinforcement learning to automate making changes to an existing model and a given desired behavioral output for the classic DEVS models (David and Syriani 2022). A mapping from the DEVS specification to a Markov Decision Process is used. Reinforcement learning, as a proximal policy optimization agent, applies one action at a time to change one instance of an atomic model to the next. These actions correspond to the elements of the classic DEVS formalism. It evaluates the output against a desired (ground truth) output. This process continues until a model that best approximates the desired behavioral output is reached. In this work, the use of RL is demonstrated for optimizing the time to the next event for a traffic light. The other elements of the atomic model (e.g., internal transition function, changing components and couplings for coupled models) remain as future work. models, simulating the models, and generating output to support model verification and simulation validation.

Using the described scenarios with human expert knowledge in the loop, a framework is proposed to generate the code based on the text prompt to simulate the inventory logistics systems (Jackson et al. 2024). This work shows that GPT-Codex (Chen et al. 2021) can be used to generate models from a simple scenario description. Python code is generated given a description of a finite capacity inventory and recurring prompts. The generated code can sequentially increase and decrease the number of entries in a numerical array list. PDEVS-LLM approach, in contrast, is aimed at generating models that can account for the structure and behavior of model components and their composition.

A framework for generating classic DEVS models is developed using LLMs (Carreira-Munich et al. 2024). This work aims to automate the specification of executable simulation models from system descriptions using agents as assistants, conversation histories, and Co-Pilot. First, one agent (named Concept Specifier) uses DEVS concepts, user prompts (iterative description of the system to be modeled), and conversation histories to generate a DEVS conceptual specification. Second, another agent (named formal specifier) uses a model specification, Python Grammar, user prompts, and conversation histories to generate a formal specification. The DEVS Copilot framework supports a process where modelers can use prompts to make changes and corrections to the conceptual and formal specifications. When the modeler is satisfied with the formal specification, a DSL parser can be used to generate PythonDEVS code. This work shows the importance of using PDEVS statecharts instead of relying on state machines used in LLMs. Generated hierarchical models can have multiple inputs and outputs.

Generative LLM (GPT-4) is also proposed for modeling DEVS-based activity modeling (Alshareef et al. 2024). DEVS-based activity is used to generate an activity model given an activity description of a sequence of actions. Prompts are used to have a sequence of actions created for the Eclipse Sirius tool. The prompts show that complications arise in creating correct behavioral models. The resultant activity diagram may be used to create and simulate DEVS models.

3 PARALLEL DEVS MODEL GENERATION

The approach outlined in (Carreira-Munich et al. 2024) is used to develop the proposed agentic framework for PDEVS. The Code, Prompts and Test cases used in this work are made available here (PDLM 2025). The objective is to leverage LLMs for generating statecharts that accurately model the behavior of atomic components. The latter stage of the pipeline focuses on storing these statecharts in a database, facilitating their manual parsing into Java code for the DEVS-Suite simulator (ACIMS 2023) . The parser developed for CoSMoS (Fard and Sarjoughian 2015) is used for generating partial executable code. The user-interaction flow illustrated in Figure 1 is described below:

- The modeler types out a system description that outlines the system, its components, and their interrelationships. This description does not need to be DEVS-specific but can be a conceptual explanation in natural language.
- PDEVS-LLM extracts distinct plausible facts for each coupled and atomic model, which are then utilized to systematically model the behavior of individual components.
- The generated statecharts undergo a human refinement loop, allowing the modeler to identify and correct any errors that arise during the behavioral modeling process.
- Following the modeler's refinement, the statecharts are converted into XML format suitable for storing in a relational database, facilitating their subsequent parsing into corresponding Java scripts (automatically or manually).

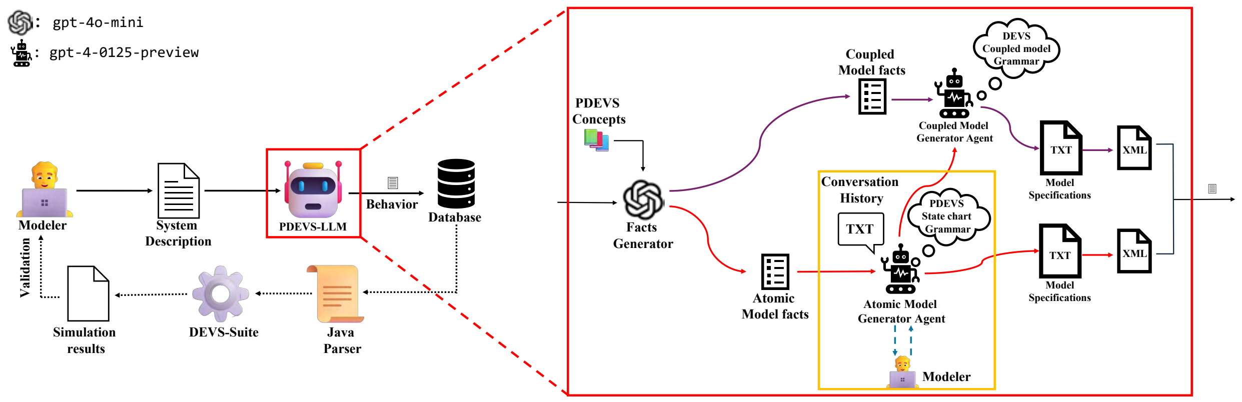


Figure 1: An illustration of the proposed PDEVS-LLM framework. The yellow box highlights the environment for refinement of PDEVS statecharts. The feedback portion of the process from Database to Modeler (shown as dotted arrows) is outside the scope of this paper.

3.1 Generative LLM Atomic and Coupled Models

The PDEVS-LLM should have the sequential steps a human expert would follow in constructing models. The workflow is illustrated in Figure 1. The proposed framework consists of three LLM agents, each dedicated to specific tasks essential for modeling the behavior of atomic and coupled models. The Facts Generator ($\mathscr{F}$) contextually includes the conceptual understanding of the PDEVS specifications as a part of its *System Prompt* and generates sets of plausible facts for coupled $\{f_{c_h}\}$ and atomic model $\{f_{a_k}\}$. The provided instructions allow $\mathscr{F}$ to present the facts for coupled models in a bottom-up manner, where low-level coupled models precede higher-level ones. This setup enables the Coupled Model Generator Agent($\mathscr{G}_{CM}$) to utilize previously generated coupled models as building blocks for constructing higher-level models. Given every PDEVS system model $\mathscr{M}$ is modular and has strict tree hierarchy structure, its set of plausible facts is $\mathscr{F}(\mathscr{M}) = \{f_{c_h}\} \bigcup \{f_{a_k}\}$.

To improve learning and minimize repeated errors, *conversation history* ($\mathscr{H}$) is used. In every iteration, the agent retains context to avoid similar mistakes in the future. The first interaction has no conversation history $\mathscr{H}_0 = \varnothing$. The impact of conversation history on the number of refinement cycles is reported in the experiment section. The Atomic Model Generator Agent ($\mathscr{G}_{AM}$) uses $\{f_{a_k}\}$ and $\mathscr{H}_\ell$ where $k \in 0, \cdots, N$ and $\ell \in 0, \cdots, I$ where there are N number of atomic models and I number of histories (modeler interactions). PDEVS statecharts $\{\mathscr{SC}_N\}$ are generated for all atomic models belonging to the system model $\mathscr{M}$. The history $\mathscr{H}_{k+1} = \mathscr{H}_k \bigcup \mathscr{M}_N$ is updated for finite numbers of atomic models N and conversation histories I. Once the statecharts for all the atomic models are generated, they are stored in a dictionary $\mathscr{SC} = \{S_{a_1}, \cdots, S_{a_N}\}$ to be used for generating coupled models.

The Coupled Model Generator Agent $\mathscr{G}_{CM}(f_c, \mathscr{SC})$ uses $\mathscr{F}$ with the grammar derived to identify the hierarchical structure and coupling among generated atomic and coupled models. The grammar defines the components, their inputs and outputs, along with external input, internal, and external output couplings. It takes in the coupled model facts and the generated atomic model behaviors and generated coupled model specifications to generate the hierarchical system model $\mathscr{M}$.

All the models (coupled and atomic) can be parsed into XML and stored in a relational database (Sarjoughian and Elamvazhuthi 2009). The `gpt-4-0125-preview` is selected to generate the model specifications, given the model's exceptional generative abilities and following instructions (Grammar). The `gpt-4o-mini` is utilized since it demonstrates comparable performance to `gpt-4-0125-preview` as it is less expensive. The pre-trained, closed-source models were used with temperatures of 0.5 and 0.7 for fact extraction and specification generation, respectively. Lower temperatures yield more deterministic outputs, while higher ones promote creativity (Peeperkorn et al. 2024).

3.2 Atomic Behavior Models as Statecharts

UML statecharts provide a standardized, powerful, and effective visual notation for defining an atomic behavior. Central to statecharts is the concept of discrete states and the transitions that occur between them. *Grammar prompting* is a simple approach for domain-specific language generation with LLMs (Wang et al. 2023). Constraints within a structured output space can often be concisely represented using a context-free grammar in Backus–Naur Form (BNF), a standard notation for defining language syntax. Given this, a grammar is developed in the Extended Backus–Naur Form (EBNF) format for the LLMs to represent the behavior of the component as a statechart. The format also allows for efficient extraction of the values corresponding to the properties and storage in XML schema.

Listing 1: PDEVS statecharts grammar in EBNF format

```
Model ::= "Model" "{" ConfluentType StateList
    TransitionList "}"
ConfluentType ::= "ConfluentType" "=" ("FIT"
    | "FET")
// FIT = First Internal Transition, FET =
    First External Transition
StateList ::= (InitialState | State |
    FinalState)+
InitialState ::= "InitialState" "{" "Name"
    "=" String "}"
State ::= "State" "{"
  "Name" "=" String
  (OutputFunction)?
"}"
FinalState ::= "FinalState" "{" "Name" "="
    String "}"
TransitionList ::= (Initialize |
    ExternalTransition | InternalTransition)+
Initialize ::=
  "Initialize" "{"
    "Action" "=" String
    "Description" "=" String
    "Source" "=" String
    "Target" "=" String
    "TimeAdvanceType" "=" ("Infinity" |
        "Update" | "Value")
    "TimeAdvanceValue" "=" Float
  "}"
ExternalTransition ::=
  "ExternalTransition" "{"
    "Action" "=" String
    "Description" "=" String
    "Guard" "=" String
    "InputPort" "=" String
    "Message" "=" String
    "Source" "=" String
    "Target" "=" String
    "TimeAdvanceType" "=" ("Infinity" |
        "Update" | "Value")
    "TimeAdvanceValue" "=" Float
  "}"
InternalTransition ::=
  "InternalTransition" "{"
    "Action" "=" String
    "Description" "=" String
    "Guard" "=" String
    "Source" "=" String
    "Target" "=" String
    "TimeAdvanceType" "=" ("Infinity" |
        "Update" | "Value")
    "TimeAdvanceValue" "=" Float
  "}"
OutputFunction ::=
  "OutputFunction" "{"
    "Action" "=" String
    "Description" "=" String
    "Guard" "=" String
    "Message" "=" String
    "OutputPort" "=" String
  "}"
```

The grammar (Listing 1) derived follows the Class-Property devised for the PDEVS statecharts editor in CoSMoS (Fard and Sarjoughian 2015). The `ConfluentType` specifies the priority of internal and external transitions when concurrent input and output events occur. An `InitialState`, `State`, and `FinalState` specify the states for the model to operate. The `OutputFuntion` attribute can be added to the State class, which includes the information about the `Message` and `OutputPort` to be used when an external event takes place. `Initialize` handles the transition to setup the initial conditions for the simulation to start. `ExternalTransition`, and `InternalTransition` are used to determine all state transitions in the model. All transitions have `Source`, `Target`, `TimeAdvanceType` and `TimeAdvanceValue` properties to store relevant information.

The `Action` property in transition classes and `OutputFunction` defines the modifications to be applied to state variables within the corresponding class. For instance, when modeling a Processor with a Queue, an array-like object named *queue* is required to store jobs before processing them in the external transition. In this case, the `Action` property should capture the operation *queue.A[job]*. The `Guard` property stores the condition to be satisfied for the transition or output function to execute. Few-shot learning (Brown et al. 2020) has emerged as an effective technique for enhancing large language model (LLM) understanding, enabling better generalization from limited examples. Motivated by this, two illustrative examples along with the corresponding grammar are introduced. Roleplay and definitions in the system prompt facilitate better model adaptation (Shanahan et al. 2023). An example component behavior modeled for **Processor with Queue** is illustrated in Listing 2. Below is the description (prompt) provided for the processor with a FIFO queuing feature:

```
We have a module named generator that sends jobs to a processor with queue.  The generator must
send only three jobs at 11th, 12th and 13th second once the simulation starts.  The initial state
of the processor is idle.  Once job signal is sent to the processor, processor takes 2 clocks
to process the job send it to the output port.  Processor will maintain the state busy during
processing and will return to idle after processing the job.  The jobs that are not ready for the
processing will wait in the queue until the processor comes out of busy state.
```

Listing 2: Behavior of the processor with queue atomic model.

```
Model {
  ConfluentType = FIT
  // States for Processor with Queue
  InitialState { Name = "start_processor" }
  State { Name = "idle" }
  State {
    Name = "busy"
    OutputFunction {
        Action = "queue.remove(),
             queue.first()"
        Description = "Output Function to send
             processed job signal after
             processing the first job in queue"
        Guard = "queue.isNotEmpty()"
        Message = "processed_job_signal"
        OutputPort = "processed_job_out"}}
  // Transitions for Processor with Queue
  Initialize {
    Action = "queue.init()"
    Description = "Initializing the Processor
         model with queue"
    Source = "start_processor"
    Target = "idle"
    TimeAdvanceType = Value
    TimeAdvanceValue = 2.0}
  ExternalTransition {
    Action = "queue.add(job), queue.first()"
    Description = "Receive job, add to queue,
         and fetch first job for processing if
         idle"
    Guard = "state=idle AND queue.isEmpty()"
    InputPort = "job_in"
    Message = "job_signal"
    Source = "idle"
    Target = "busy"
    TimeAdvanceType = Value
    TimeAdvanceValue = 2.0}
  ExternalTransition {
    Action = "queue.add(job)"
    Description = "Receive job and add to queue
         if busy"
    Guard = "state=busy OR NOT queue.isEmpty()"
    InputPort = "job_in"
    Message = "job_signal"
    Source = "busy"
    Target = "busy"
    TimeAdvanceType = Update
    TimeAdvanceValue = 0.0}
  InternalTransition {
    Action = ""
    Description = "Check if queue is empty to
         go idle"
    Guard = "queue.isEmpty()"
    Source = "busy"
    Target = "idle"
    TimeAdvanceType = Value
    TimeAdvanceValue = 2.0}
   InternalTransition {
    Action = "queue.remove(), queue.first()"
    Description = "Process next job in queue if
       not empty after current job"
    Guard = "NOT queue.isEmpty()"
    Source = "busy"
    Target = "busy"
    TimeAdvanceType = Value
    TimeAdvanceValue = 2.0}}
```

3.3 Hierarchical Coupled Models

Real-world systems are organized hierarchically, enabling the decomposition of complex behavior into smaller, more manageable modules. Such hierarchical structuring facilitates abstraction by allowing modelers to focus on high-level system behavior while encapsulating low-level implementation details. The modelers often follow a bottom-up fashion while creating the coupled models in the system. Coupled Model Generator Agent imitates the same while generating the coupled models for the system. The specific grammar (Listing 3) and syntactic rules are developed to enable generating multi-level model hierarchies.

Listing 3: PDEVS Coupled model grammar in EBNF format

```
Model ::= "Coupled Model"
ModelInPortList ::= (ModelInPort)*
ModelInPort ::= "ModelInPort" "{" "Name" "="
     String "}"
ModelOutPortList ::= (ModelOutPort)*
ModelOutPort ::= "ModelOutPort" "{" "Name"
     "=" String "}"
ComponentList ::= (Component)+
Component ::= "Component" "{"
   "Name" "=" String
   InPortList
   OutPortList
"}"
InPortList ::= (InPort)*
InPort ::= "InPort" "{" "Name" "=" String "}"
OutPortList ::= (OutPort)*
OutPort ::= "OutPort" "{" "Name" "=" String
     "}"
InternalCouplingList ::= (Coupling)+
ExternalInputCouplingList ::= (Coupling)+
ExternalOutputCouplingList ::= (Coupling)+
Coupling ::= "Coupling" "{"
   "M1Name" "=" String
   "M1Port" "=" String
   "M2Name" "=" String
   "M2Port" "=" String
"}"
```

`ModelInPort` and `ModelOutPort` classes are used to store the values of the coupled model ports. `ComponentList` stores the atomic or coupled models associated with the coupled model. The `InternalCouplingList` is used to store `Coupling` instances that represent connections between components (internal). `ExternalInputCouplingList`, and `ExternalOutputCouplingList`, are used to store `Coupling` instances that represent connections from the coupled model's input ports to component input ports, and from component output ports to the coupled model's output ports, respectively. A coupled system description consisting of one Generator and one coupled Processor-Sensor system is devised. The grammar in Listing 3 is used to generate the coupled model depicted in Listing 4.

Listing 4: Processor-Sensor coupled model specifications (Hierarchy Level 2)

```
Coupled Model {
 Name = "processor-sensor"
 ModelInPortList{ ModelInPort { Name =
     "start" } }
 ModelOutPortList{ ModelOutPort { Name =
     "stop" } }
 Component {
  Name = "processor"
  InPortList { InPort { Name = "in" }}
  OutPortList { OutPort { Name = "out_1" }}}
 Component {
  Name = "sensor"
  InPortList { InPort { Name = "in" }}
  OutPortList { OutPort { Name = "out" }}}
 InternalCouplingList{
  Coupling {
   M1Name = "processor"
   M1Port = "out_1"
   M2Name = "sensor"
   M2Port = "in"}}
 ExternalInputCouplingList{
  Coupling {
   M1Name = "processor-sensor"
   M1Port = "start"
   M2Name = "processor"
   M2Port = "in"}}
 ExternalOutputCouplingList{
  Coupling {
   M1Name = "sensor"
   M1Port = "out"
   M2Name = "processor-sensor"
   M2Port = "stop"}}}
```

4 EXPERIMENTAL RESULTS AND ANALYSIS

Real-world systems have components that have complex behavior traits, such as handling and storing multiple external events. The following is an illustrative example.

4.1 Modeling Atomic Model Behavior through Generator-Processor-Sensor System Description

Generator: The Generator components produce jobs. Each component can incorporate multiple behaviors. It can produce jobs at particular time instances or at a regular frequency. Each generator halts generating jobs when it receives a stop input.
Processor: The Processor component handles processing the jobs incoming from the multiple sources separately or simultaneously. Similar to the real world components, they handle the jobs in a queue and process them in a FIFO fashion. They can also differentiate the jobs coming from multiple sources and output them to the corresponding outputs.
Sensor: The Sensor component is for monitoring the processor outputs. In real-world assembly lines, a halt condition is needed to stop production when a certain criterion is met. Upon satisfying the criterion, Sensor sends the stop output event to Generators.

The following is a description of a modular, hierarchical PDEVS model that has three atomic models and two coupled models:

```
The system model has two generators, a processor, and two sensors with feedforward and feedback
couplings.
Generator_A has one input port and one output port and produces jobs at each 3 seconds periodically
till stop signal is received on the stop port.  On receiving the stop signal, Generator_A will stop
generating the jobs and go into the "passive" state.  The generated Jobs should be like JobA1,
JobA2, ...
Generator_B has one input port stop and two output ports (out_1, and out_2).  The jobs will be
generated on both the ports at each 6 seconds periodically till stop signal is received.  On
receiving the stop signal, Generator_A will stop generating the jobs and go into the "passive"
state.  The generated jobs from out_1 should look like JobB11, JobB12, ...The generated jobs from
out_2 should look like JobB21, JobB22, ...
The Processor will have two input ports (in_1, and in_2) and two output ports (out_1, out_2) to
take in the jobs.  There is a single queue which will accommodate the incoming jobs.  There is no
priority over the jobs coming at the particular input port to be processed first.  The processor
will take 2 seconds to process a job.  It will be initiated in idle state and will maintain busy
state while processing and will go back to idle if there are no jobs left in the queue.  The jobs
coming from Generator_A should go to out_1 after processing.  The jobs from Generator_B should
go to out_2 after processing.
The Sensor_A will receive the processed jobs from the out_1 of the processor and will keep the
count.  Once the 5 jobs are processed, the stop signal on the output port will stop Generator_A.
The Sensor_B will receive the processed jobs from the out_2 of the processor and will keep the
count.  Once the 5 jobs are processed, the stop signal on the output port will stop Generator_B.
The output port of the Generator_A will be connected to the in_1 of the processor.  The out_1 of
the Generator_B will be connected to the in_1 of the processor and out_2 of the Generator_B will be
connected to the in_2 of the processor.  The out_1 of the processor will be connected to the in port
of Sensor_A. The output port of the Sensor_A will be connected to the in port of Generator_A. The
out_2 of the processor will be connected to the in port of Sensor_B. The out port of the Sensor_B
will be connected to the in port of Generator_B.
```

The above System Description is used as the prompt to generate the Generator-Processor-Sensor system. The prompt is engineered to specify the structure and behavior of each component. It describes the coupling to be followed to create the hierarchy of the system. The statecharts (Table 1) for all atomic models are generated similarly to Listing 2.

The basic processor in Section 2.2 is extended with a FIFO queue. This processor can receive/store multiple jobs simultaneously as well as identify the type of job before processing, followed by outputting the processed jobs on an output port. From Table 1, it is evident that the proposed Agentic framework supports generating the internal, external, output, and initialization elements of PDEVS. The specification

Component	Transition	Source State	Target State	Guard Condition	Action	Output Function	Time Advance
Generator$_A$	Initializing	start	active	–	–	–	3.0
	Internal	active	active	State == active	job$^{++}_{COUNT}$	Port$_{OUT1}$ = "Job$_{A/BX}$"	3.0
	External	active	passive	State == active, Port$_{STOP}$ == "stop"	–	–	Infinity
Processor	Initializing	start	idle	–	queue.init()	–	Infinity
	External	idle	busy	Port$_{in1}$ == Job$_{AX}$/Job$_{B1X}$ Port$_{IN2}$ == Job$_{B2X}$	queue.add(Job..)	–	2.0
	External	busy	busy	Port$_{IN1}$ == Job$_{AX}$/Job$_{B1X}$ Port$_{IN2}$ == Job$_{B2X}$	queue.add(Job..)	–	2.0
	Internal	busy	busy	queue.length() != 0	queue.remove(Job..)	Port$_{OUT1}$ = "Job$_{AX}$/Job$_{B1X}$" or Port$_{OUT2}$= "Job$_{B2X}$"	2.0
	Internal	busy	idle	queue.length() == 0	–	–	Infinity
Sensor$_A$	Initializing	start	monitoring	–	processed$_{COUNT}$ = 0	–	Infinity
	External	monitoring	monitoring	Port$_{IN}$ == "Job$_{XX}$"	processed$^{++}_{COUNT}$	–	0.0
	Internal	monitoring	monitoring	processed$_{COUNT}$ == 5	–	Port$_{OUT}$ = "stop"	0.0

Table 1: Generated PDEVS statecharts in tabular form are for the Generator$_A$, Processor, and Sensor$_A$ components. The Generator$_B$ is similar to Generator$_A$. Sensor$_B$ is the same as Sensor$_A$.

for these functions complies with the PDEVS statecharts, enabling modeling involving complex behavior (see Section 2.2). It can be noted that the state variable phase is only needed for state transition and output functions. This processor with FIFO queuing can have other statecharts.

4.2 Refinement Analysis

The DEVS Copilot was tested using seven different configurations of Controller, Light, and Sensor components, each having a single input and a single output. For comparison purposes, the same configurations are used and tested using PDEVS-LLM. Only the histories of the atomic component models are used. Individual system descriptions used in DEVS Copilot do not have dependency on the previous systems in contrast to DEVS Copilot. The original descriptions for these systems are available (Carreira-Munich et al. 2024). The prompts used for the comparison study (see Table 2) are provided below:

```
System-1:  We have a module named controller that sends on and off signals to a light.  The
controller must send these commands after every second.  The initial state of the light is off and
the controller will auto-start the cycle 10 seconds after the simulation start.  Once started,
toggling should never end.
--------------------------------------------------------------------------------
System-2:  We have a module named controller that sends on and off signals to five identical
lights.  The controller must send these commands after every second.  The initial state of the
lights is off and the controller will auto-start the cycle 10 seconds after the simulation start.
Once started the toggling should never end.  Even though the light component is to be repeated five
times, give the facts for each light by naming this atomic component as light1, light2, light3....
--------------------------------------------------------------------------------
System-3:  We need a controller-light coupled model that can be used as a single unit.  The atomic
module controller sends on and off signals to it corresponding light.  The initial state of the
lights is off and the controllers will auto start the cycle 10 seconds after the simulation start.
Of this controller-light abstraction, create 5 pairs.  Although the components are identical,
generate all the coupled facts by naming them 'coupled1', 'coupled2', and so on.  Similarly for
the atomic components, generate controller1, controller2...., light1, light2,...  Modify the
controllers to have 1,2,3,4,5 seconds of frequency in generating the on and off signals.
--------------------------------------------------------------------------------
System-4:  We have five controller modules, each sending on and off signals to a single light with
the frquencies of 1,2,3,4,5 seconds.  The initial state of the light is off and the controllers
will auto start the cycle 10 seconds after the simulation starts.  Although the components are
identical, generate all the atomic facts by naming them 'controller1', 'controller2',..  and so
on.
--------------------------------------------------------------------------------
System-5:  We have a module named controller that sends on and off signals to a light.  The
controller must send these commands after every second.  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.  We add a sensor to this setup to know if the light is on or off.
--------------------------------------------------------------------------------
```

System ID	DEVS Copilot		PDEVS-LLM	
	A	**B**	**C**	**D**
1	2	0*	Controller: 1; Light: 1	Controller: 1; Light: 0
2	1	2*	Controller: 4; Lights: 7	Controller: 2; Lights: 1
3	1	0*	Controllers: 1; Light: 0	Controller: 2; Lights: 1
4	0	1*	Controllers: 4; Light: 3	Controllers: 1; Light: 0
5	2	1*	Controller: 1; Light: 4; Sensor: 4	Controller: 2; Light: 1; Sensor: 0
6	4	0†	Controller: 4; Light: 2; Sensor: 3	Controller: 3; Light: 0; Sensor: 2
7	1	0♦	Controllers: 4; Lights: 1; Sensor: 3	Controllers: 1; Lights: 2; Sensor: 0

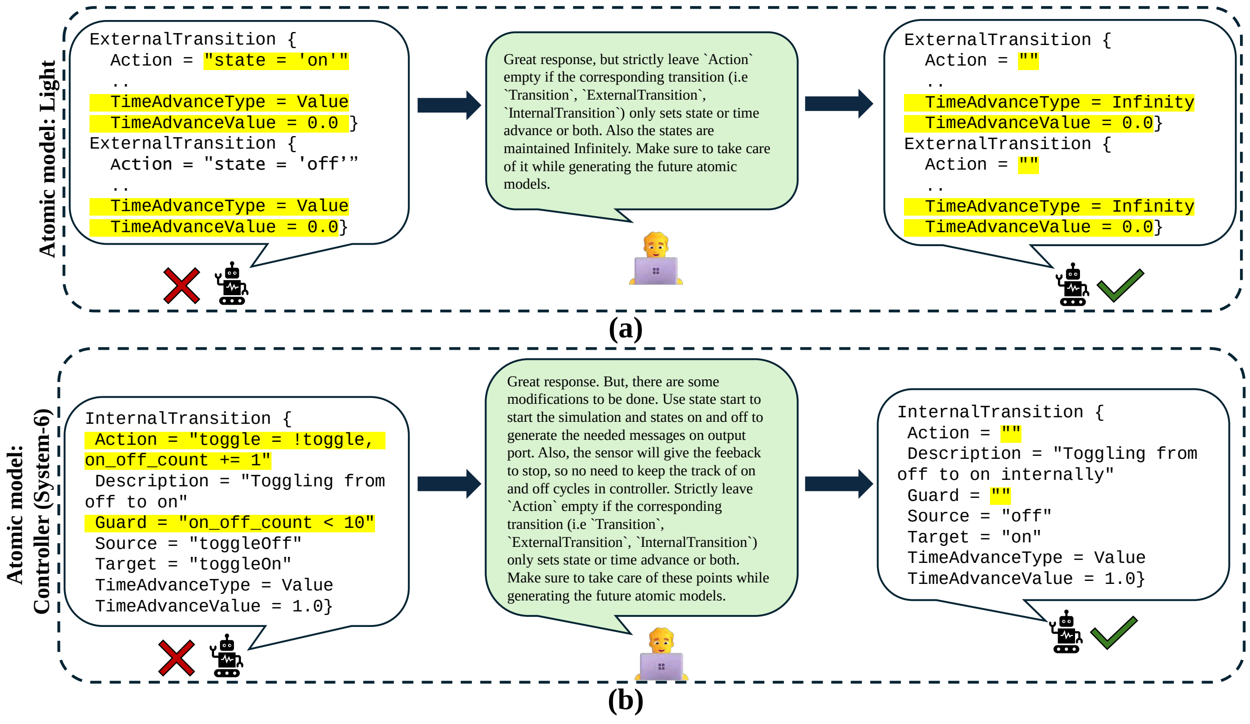
Table 2: Refinement cycles comparison between DEVS Copilot and PDEVS-LLM. *, †, and ♦ stand for Base Systems 1, 5, and 6, respectively. Columns **A** and **C** do not use history while columns **B** and **D** do.

```
System-6:  We have a module named controller that sends on and off signals to a light.  The
controller must send these commands after every second.  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.  We add a sensor to this setup to know if the light is on or off.
The sensor should send the feedback 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.
-----------------------------------------------------------------------------------------
System-7:  We have three controller modules, each sending on and off signals to three lights
individually with the frquencies of 1,2,3 seconds.  The initial state of the lights is off and the
controllers will auto-start the cycle 10 seconds after the simulation start.  We add a sensor to
this setup to keep the track of total on-off cycles irrespective of the light.  The sensor should
send the feedback to all the controllers, telling them to stop toggling the light and simply turn
it off.  This should happen after 10 on-off light cycles are detected, the sensor is the component
responsible of knowing when that threshold is reached.
```

Table 2 demonstrates that incorporating conversation history reduces the number of refinement cycles in the DEVS Copilot and PDEVS-LLM frameworks. The concept and use of *history* are different. In DEVS Copilot, a conversation history is specific to one component of the system at a time. For the PDEVS statecharts, the history is the accumulation of all previous histories for a system. The history contains identified mistakes and prompt modifications belonging to each of the atomic components of the system. The Red color highlights that DEVS Copilot produced the erroneous output for the corresponding systems (even after refinements). The Blue color states the inability of DEVS Copilot to generate coupled models. It should be noted that, unlike DEVS Copilot, PDEVS-LLM does not generate executable code. The PDEVS statecharts can be automatically transformed into code snippets for the DEVS-Suite simulator (Fard and Sarjoughian 2015). The state transition functions should be manually completed except when a model has a finite statespace with primitive values and the time base is restricted to have only discrete values. It should be noted Copilot should not be expected to write correct and complete code ready for simulation.

In PDEVS-LLM, errors for atomic Models mainly stem for two reasons: (1) misinterpretation of the properties in the system prompt, resulting in incorrect value predictions (syntactical mistakes) and (2) confusion regarding behavioral logic, leading to incorrect generation of state transitions and output functions. Figure 2 illustrates the examples of these errors and modifications to correct them.

Figure 2 (a) describes the syntactical mistake made in `Action` and `TimeAdvanceType` parts for the Light component used in all systems. The `Action` property, in particular, is not intended to store the logic for state or time advance changes such as in the `Controller` component. This mistake appears consistently across all seven example systems. In another example (not shown due to space constraints), the *Light* component's on/off state is supposed to remain unchanged (i.e., `TimeAdvanceValue = Infinity`) unless an external event is received. For the Light component, the value of the `TimeAdvanceValue` should be set to 0.0 when an input value `on` is received on the `signal-in` input port.



```
ExternalTransition {
  Action = "state = 'on'"
  ..
  TimeAdvanceType = Value
  TimeAdvanceValue = 0.0 }
ExternalTransition {
  Action = "state = 'off'"
  ..
  TimeAdvanceType = Value
  TimeAdvanceValue = 0.0}
```

Great response, but strictly leave `Action` empty if the corresponding transition (i.e `Transition`, `ExternalTransition`, `InternalTransition`) only sets state or time advance or both. Also the states are maintained Infinitely. Make sure to take care of it while generating the future atomic models.

```
ExternalTransition {
  Action = ""
  ..
  TimeAdvanceType = Infinity
  TimeAdvanceValue = 0.0}
ExternalTransition {
  Action = ""
  ..
  TimeAdvanceType = Infinity
  TimeAdvanceValue = 0.0}
```

```
InternalTransition {
 Action = "toggle = !toggle,
on_off_count += 1"
 Description = "Toggling from
off to on"
 Guard = "on_off_count < 10"
 Source = "toggleOff"
 Target = "toggleOn"
 TimeAdvanceType = Value
 TimeAdvanceValue = 1.0}
```

Great response. But, there are some modifications to be done. Use state start to start the simulation and states on and off to generate the needed messages on output port. Also, the sensor will give the feeback to stop, so no need to keep the track of on and off cycles in controller. Strictly leave `Action` empty if the corresponding transition (i.e `Transition`, `ExternalTransition`, `InternalTransition`) only sets state or time advance or both. Make sure to take care of these points while generating the future atomic models.

```
InternalTransition {
 Action = ""
 Description = "Toggling from
off to on internally"
 Guard = ""
 Source = "off"
 Target = "on"
 TimeAdvanceType = Value
 TimeAdvanceValue = 1.0}
```

Figure 2: Refinement for errors stemming from **(a)** syntactical mistakes and **(b)** behavioral logic.

The second type of error is due to the confusion in behavioral logic (refer to Figure 2 (b)) as illustrated for System-6. The system prompt describes that the `Sensor` component to be responsible for keeping the count of on/off cycles and trigger the feedback `stop` output to the `Controller` once the desired threshold is reached. Instead, this behavior should be assigned to the `Sensor` component.

5 CONCLUSION

In this research, the PDEVS-LLM agentic framework using PDEVS statecharts is introduced. By leveraging the statecharts grammar along with PDEVS atomic and coupled model specifications, the framework can assist in generating behavioral parallel atomic and structural hierarchical coupled models and transforming them into XML schemas. These can be stored and transformed into partial code that can be manually completed for simulators supporting the PDEVS formalism. The generated statecharts should be verified for errors since they contain mistakes similar to those human modelers make. Errors stem from plausible facts and the history of the prior statecharts generated from the system's textual description and changes made by the human modelers. The PDEVS statecharts struggle with handling complex timing, such as a generator dispatching outputs on multiple ports at different frequencies. Future research on PDEVS includes enabling LLMs to model multiple concurrent inputs and outputs, supporting events that can occur at arbitrary time instances, and developing the means to formally verify model correctness.

ACKNOWLEDGMENTS

This research is funded by Intel Corporation, Chandler, Arizona, USA.

REFERENCES

ACIMS 2023. "DEVS-Suite Simulator, version 7.0". https://acims.asu.edu/devs-suite/, accessed 15th January 2025.

Alshareef, A., N. Keller, P. Carbo, and B. P. Zeigler. 2024. "Generative AI with Modeling and Simulation of Activity and Flow-Based Diagrams". In *Simulation Tools and Techniques*, 95–109. Cham: Springer Nature Switzerland.

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, *et al*. 2020. "Language Models are Few-Shot Learners". In *Advances in Neural Information Processing Systems*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Volume 33, 1877–1901: Curran Associates, Inc.

Carreira-Munich, T., V. Paz-Marcolla, and R. Castro. 2024. "DEVS Copilot: Towards Generative AI-Assisted Formal Simulation Modelling based on Large Language Models". In *2024 Winter Simulation Conference (WSC)*, 2785–2796 https://doi.org/10.1109/WSC63780.2024.10838994.

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*.

David, I., and E. Syriani. 2022. "DEVS Model Construction As A Reinforcement Learning Problem". In *2022 Annual Modeling and Simulation Conference (ANNSIM)*, 30–41.

Fan, L., L. Li, Z. Ma, S. Lee, H. Yu, and L. Hemphill. 2024, October. "A Bibliometric Review of Large Language Models Research from 2017 to 2023". *ACM Trans. Intell. Syst. Technol.* 15(5).

Fard, M. D., and H. S. Sarjoughian. 2015. "Visual and Persistence Behavior Modeling for DEVS in CoSMoS". In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative MS Symposium*, DEVS '15, 227–234. San Diego, CA, USA: Society for Computer Simulation International.

Gao, C., X. Lan, N. Li, Y. Yuan, J. Ding, Z. Zhou, *et al*. 2024. "Large language Models Empowered Agent-based Modeling and Simulation: A Survey and Perspectives". *Humanities and Social Sciences Communications* 11(1):1–24.

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.

Jackson, I., M. J. Saenz, and D. Ivanov. 2024. "From Natural Language to Simulations: Applying AI to Automate Simulation Modelling of Logistics Systems". *International Journal of Production Research* 62(4):1434–1457.

Capocchi L. 2018. "PythonPDEVS". https://github.com/capocchi/PythonPDEVS, accessed 18th February 2018.

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*.

Liu, X., H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, *et al*. 2023. "AgentBench: Evaluating LLMs as Agents". *arXiv preprint arXiv: 2308.03688*.

PDLM 2025. "Parallel DEVS LLM Modeler". https://github.com/comses/PDLM, accessed 4th August 2025.

Peeperkorn, M., T. Kouwenhoven, D. Brown, and A. Jordanous. 2024. "Is Temperature the Creativity Parameter of Large Language Models?". *arXiv preprint arXiv:2405.00492*.

Sarjoughian, H. S., and V. Elamvazhuthi. 2009. "CoSMoS: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation". In *Proceedings of the 2nd international conference on simulation tools and techniques*, 1–9.

Shanahan, M., K. McDonell, and L. Reynolds. 2023. "Role play with Large Language Models". *Nature* 623(7987):493–498.

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.

Wang, B., Z. Wang, X. Wang, Y. Cao, R. A. Saurous, and Y. Kim. 2023. "Grammar Prompting for Domain-Specific Language Generation with Large Language Models". In *Advances in Neural Information Processing Systems*, edited by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Volume 36, 65030–65055: Curran Associates, Inc.

Zeigler, B. P., A. Muzy, and E. Kofman. 2018. *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. 3rd ed. Academic press.

AUTHOR BIOGRAPHIES

VAMSI KRISHNA VASA is a M.Sc student in the Computer Science program in the School of Computing and Augmented Intelligence (SCAI) at Arizona State University (ASU), Tempe, AZ, USA. He can be reached at vvasa1@asu.edu.

HESSAM S. SARJOUGHIAN is an Associate Professor of Computer Science and Computer Engineering in the School of Computing and Augmented Intelligence (SCAI) at Arizona State University (ASU), Tempe, Arizona. His research interests include model theory, poly-formalism modeling, machine learning, collaborative modeling, simulation for complexity science, and M&S frameworks/tools. He is the co-director of the Arizona Center for Integrative Modeling and Simulation https://acims.asu.edu. He can be contacted at hessam.sarjoughian@asu.edu.

EDWARD J. YELLIG is the director of Operational Decisions Support Technology at Intel Corporation. He has been with Intel for 26 years and has a Ph.D. in Operations Research with an emphasis on discrete event modeling of large-scale systems. His focus has been on developing fab models for determining capital requirements and is also responsible for the real-time digital twin tactical models. He can be contacted at edward.j.yellig@intel.com.