Solution-oriented Agent-based Models Generation with Verifier-assisted Iterative In-context Learning

Tong Niu
Center for Brain-Inspired Computing
Research, Tsinghua University
Beijing, China
nt20@mails.tsinghua.edu.cn

Weihao Zhang
Lynxi Technologies Co. Ltd
Beijing, China
zwh18@mails.tsinghua.edu.cn

Rong Zhao
Center for Brain-Inspired Computing
Research, Tsinghua University
Beijing, China
r_zhao@mail.tsinghua.edu.cn

ABSTRACT

Agent-based models (ABMs) stand as an essential paradigm for proposing and validating hypothetical solutions or policies aimed at addressing challenges posed by complex systems and achieving various objectives. This process demands labor-intensive endeavors and multidisciplinary expertise. Large language models (LLMs) encapsulating cross-domain knowledge and programming proficiency could potentially alleviate the difficulty of this process. However, LLMs excel in handling sequential information, making it challenging for analyzing the intricate interactions and nonlinear dynamics inherent in ABMs. Additionally, due to the lack of self-evaluation capability of LLMs, relying solely on LLMs is insufficient to effectively accomplish this process. In this paper, we present SAGE, a general solution-oriented ABM generation framework designed for automatic modeling and generating solutions for targeted problems. Unlike approaches reliant on expert handcrafting or resource-intensive neural network training, SAGE establishes a verifier-assisted iterative in-context learning process employing large language models (LLMs) to leverages their inherent cross-domain knowledge for tackling intricate demands from diverse domain scenarios. In SAGE, we introduce an semi-structured conceptual representation expliciting the intricate structures of ABMs and an objective representation to guide LLMs in modeling scenarios and proposing hypothetical solutions through in-context learning. To ensure the model executability and solution feasibility, SAGE devises a two-level verifier with chain-of-thought prompting tailored to the complex interactions and non-linear dynamics of ABMs, driving the iterative generation optimization. Moreover, we construct an evaluation dataset of solution-oriented ABMs from open sources. It contains practical models across various domains, completed with scenario descriptions and executable agent-based solutions. Evaluations by various LLMs demonstrate that SAGE leads to an average improvement of 18.7% in modeling quality and 38.1% in solution generation effectiveness. This work advances our understanding and ability in tackling complex real-world challenges across diverse domains through the application of ABM methodologies.

Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), N. Alechina, V. Dignum, M. Dastani, J.S. Sichman (eds.), May 6 – 10, 2024, Auckland, New Zealand. © 2024 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

KEYWORDS

Iterative in-context learning; Solution-oriented agent-based modeling; Automatic verification and generation; Chain-of-thought prompting; Large language models

ACM Reference Format:
Tong Niu, Weihao Zhang, and Rong Zhao. 2024. Solution-oriented Agent-based Models Generation with Verifier-assisted Iterative In-context Learning. In *Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2024), Auckland, New Zealand, May 6 – 10, 2024*, IFAAMAS, 9 pages.

1 INTRODUCTION

Agent-based models (ABMs) provide a bottom-up perspective for exploring complex systems [17], such as societies [27], ecosystems [7] and financial markets [38]. By modeling and simulating actions and interactions of autonomous agents within an artificial environment, ABMs can replicate real-world phenomena, validate hypotheses, uncover emergent behaviors, and formulate solutions or policies for specific problems. In the process of formulating solutions using ABMs, practitioners typically begin by modeling problem scenarios and subsequently simulating the hypothetical solutions proposed by experts to verify their effectiveness [22, 41]. This process poses significant challenges for practitioners [43], including the need for proficiency in abstract modeling to capture essential elements, structures and dynamics of the system to simplify real-world scenarios into an agent-based model. It also requires exceptional programming skills to translate the model into computational code. Most importantly, it demands a solid foundation of domain knowledge that entails a deep understanding of the problem context.

Leveraging advancements in computing methods [20, 21], substantial efforts are dedicated to automating the aforementioned modeling and solution generation process. Some initiatives apply intelligent algorithms, such as reinforcement learning [34], to automatically generate strategies [18, 46]. However, these approaches demand a prolonged learning processes reliant on explicit reward design or vast amounts of data. Some works make innovations by proposing domain-specific languages [37] or abstracting modeling principles [29] to simplify the programming complexity of models. Recently, large language models (LLMs) have shown significant progress in various tasks [16, 36], such as code generation [31] and question answering [3]. Through extensive pre-training on diverse and abundant data, LLMs acquire the ability to generalize knowledge and apply it across various domains, positioning them as potentially valuable tools for the automated generation of solution-oriented ABMs. However, since LLMs are primarily designed to model natural languages, they excel in processing sequential information and linear causality [8]. When applied directly to generate, analyze, and enhance ABMs, LLMs may encounter limitations due to their inherent design constraints in dealing with non-linear dynamics and intricate relationships among heterogeneous agents and the environment of ABMs. Besides, LLMs lack the self-checking and self-correction abilities for generated content. Therefore, ABMs generated by LLMs may contain factual inaccuracies, logical inconsistencies, and oversimplified intricacies of the real-world systems they aim to represent. Balancing the desire to streamline the solving process using LLMs with the need to ensure the executability and efficacy of the solution-oriented ABMs poses a formidable challenge.

To address these challenges, we present SAGE (Solution-oriented Agent-based models GEneration), a framework designed to leverage the in-context learning ability of LLMs for generating executable and verifiable agent-based solutions. SAGE comprises two stages: **"Modeling" stage** that uses a conceptual representation coupled with few-shot prompting techniques [32] to help LLMs comprehend the holistic structure and dynamics of the problem, ultimately facilitating the generation of executable ABMs that replicate the given problem scenario. **"Solving" stage** that employs iterative chain-of-thought (CoT) [42] prompting with a two-level verifier that incorporates an objective representation to automatically generate and optimize effective solutions based on the above executable ABMs.

In SAGE, verifier-level1 identifies "compilation errors" and "lacking details" to ensure the executability and integrity of the ABM program. Meanwhile, verifier-level2 guide the iterative generation of CoT prompts, optimizing solutions by comparing objective criteria and simulation results. The CoT prompting breaks down the solution generation process into three steps: relations extraction, cause analysis, and solution proposal, enabling LLMs to formulate targeted solutions by comprehensively understanding the problem and the model. Both the conceptual representation (describing the problem scenario) and the objective representation (specifying the desired solution effects) serve as inputs to the LLM. These representations strike a balance between being semi-natural language (NL) for expression-diverse and user-friendly interface, and semi-structured for program accuracy and in-context learnability. By employing this approach, SAGE empowers LLMs to effectively organize intricate interactions within ABMs and leverage their inherent domain knowledge to propose efficient solutions. Furthermore, we have constructed an evaluation dataset comprising solution-oriented ABMs from open-source projects. Unlike existing code generation evaluation datasets, which primarily consist of partial code fragments accompanied by NL descriptions [25, 40], our dataset offers comprehensive scenario descriptions along with their ABM implementations. It encompasses real-world problem instances from various fields and their corresponding ABM-based solutions. The evaluation results of using SAGE with various state-of-the-art LLMs on this dataset demonstrate a massive improvement on generated ABMs' quality and problem-solving effectiveness, when compared to approaches that do not utilize the SAGE framework. The main contributions of our work are as follows:

(1) SAGE: a framework that enables the automatic generation of executable and verifiable agent-based solutions through "Designing" and "Solving" stages. (2) The semi-structured conceptual and objective representations to formalize the ABM initialization and targets while maintaining the NL's expressivity. (3) A two-level verifier and CoT prompting method to streamline LLMs in generating effective solutions through iterative in-context learning. (4) A solution-oriented ABMs dataset to evaluate the quality of modeling and solution generation across multiple domains.

2 RELATED WORK

2.1 Automatic Design and Generation of ABM

The field of data and computational science has witnessed remarkable progress over the past decade, leading to a shift in the design and construction of ABMs from manual to automated approaches. This transition has empowered better exploration of data and expression of models. In terms of model structure generation, [30] proposes the MAGIC algorithm, which converts the raw observational data to a sequential compressed Markov decision process as agents' behavior and decision structures. Through a meta-programming system, [43] can automatically generate executable ABM codes from an "Overview, Design concepts, and Details" (ODD) [11] description. [10] provides an evolvable approach to generate interpretable agent logic from scratch through program synthesis [12] and genetic programming [19], which endow users with transparency. Regarding the generation of agents' decisions, [46] proposes a two level deep multi-agent reinforcement learning framework for automatic design of taxation policy. [18] proposes a democratic AI capable of designing social mechanisms based on the majority preferences of humans through a human-in-loop reinforcement learning pipeline. These automatic generation methods often require extensive domain knowledge or long-term learning process based on massive data. Besides, some methods primarily focus on certain scenarios, lacking generality. Our approach overcomes these limitations with the ability of LLMs in efficiently handling multi-domain generation tasks, which is acquired during the large-scale pre-training phase.

2.2 LLMs for Code generation

LLMs have achieved striking performance in a wide range of fields, including code generation tasks. Typically, these programming LLMs are developed by conducting self-supervised language modeling tasks on large unlabelled program language corpora followed by simple modifications and affordable fine-tuning. Specifically, three transformer architectures are widely developed: BERT [4], BART, and GPT [33]. As for BERT, CodeBERT [6] is pre-trained on millions of program functions across six programming languages. GraphCodeBERT [13], ContraBERT [24] are improved models that concern the inherent structure of code and enhance the robustness of CodeBERT. In terms of BART, PLBART [1] undergoes pre-training using a vast dataset comprising Java and Python functions along with their corresponding NL text, through denoising auto-encoding techniques. CommitBART [23] pre-trains BART using data collected from GitHub commits. Regarding GPT, CodeGPT [25] performs pre-training on Python and Java datasets sourced from the CodeSearchNet [15]. CodeX [2] conducts fine-tuning of GPT-3, which automatically fixed 28.8% of programming problems in HumanEval datasets and is underlying technology behind GitHub Copilot [9]. These works have performed well on code snippet generation tasks. However, their efficacy tends to diminish in tasks like ABM programming that involve heterogeneous agents and intricate interaction patterns. To address this limitation, we augment LLMs' in-context learning by incorporating logically coherent and semi-structured representations.

2.3 In-context Learning

LLMs are capable of in-context learning (ICL) to learn tasks and generate desired output given only a few demonstration examples [5]. ICL utilizes the analogy ability of the LLM and augments their task-specific working memory within the context, thereby influencing the likelihood of generating appropriate responses. Therefore, ICL can be applied during the inference procedure of a well-trained LLM, obviating the need for additional training. The organization of demonstrations within prompts, including their order and format significantly impacts ICL performance [26, 45]. Many prompting methods have been proposed. [14] and [47] adopt manual construction such as providing template-based prompts and few-shot prompting. [48] and [44] leverage LLMs for automatic prompt generation. The CoT approach introduces intermediate reasoning steps to guide LLMs in mastering complex reasoning[42]. CoT has been widely used and developed since it was proposed [32]. Some researchers employ multi-stage ICL for CoT prompting and design CoT demonstrations for each step [39, 47]. [28] and [35] introduce an optimizer or verifier to calibrate the generation result through iterative generation and optimization procedures. ICL has found applications in various domains, including visual tasks [14], language reasoning [32], and task planning [35]. However, there is limited research on utilizing ICL to guide LLMs to generate high quality ABMs.

3 SAGE

Architecture and Workflow SAGE divides the solution generation process into two stages, as shown in Figure 1. The initial stage, termed the **"Modeling" stage**, aims to generate an executable ABM to simulate a given problem scenario. During this stage, the LLM creates the ABM program based on the conceptual representation of the scenario provided by users. This semi-structured representation explicitly depicts the intricate interactions and dynamic events within ABMs through its structured components, facilitating analysis by LLMs. Besides, it enables users to describe scenarios using more expressive natural language. The generated ABM then undergoes automatic verification for both executability and integrity by the verifier-level1. The LLM iteratively rectifies the program based on the verification results until a correct ABM is obtained. Such verifier-assisted iteration compensates for the self-checking deficiencies in LLMs, avoiding execution errors and oversimplifications.

Subsequently, the process transitions to the **"Solving" stage** that focuses on generating effective solutions for given problems. In this stage, users specify the target problem and desired effects or constraints of the solution using objective representations. (Both conceptual and objective representations are concatenated with examples to form input prompts for ICL.) Then, verifier-level2 is designed to utilize LLMs to generate verification functions based on the objective representations. These functions are used to verify whether simulation results meet the specific objectives, which drives the optimization of solutions and compensates for the self-evaluation deficiencies in LLMs. Based on the verification results, verifier-level2 provides CoT instructions that guide LLMs in generating solutions step by step, which include relations extraction, cause analysis, and solution proposal. This CoT design guides LLMs in recognizing the holistic structure of the current ABM and subsequently formulating more specific solutions based on relevant elements, thereby mitigating the limitations of LLMs in deducing the complex interaction dynamics of ABMs solely from static simulation results. The generated solutions are integrated back into the existing ABM and undergo further iterative rectification using verifier-level1 until the ABM with new solutions is executable and comprehensive. This process continues iteratively between verifier-level1 and verifier-level2 until the desired objectives are met. It is worth noting that these two stages can be decoupled and independently utilized depending on research purposes, allowing flexibility in applications. The pseudocode outlining the "Modeling" stage and "Solving" stage can be found in Algorithm 1 and Algorithm 2.

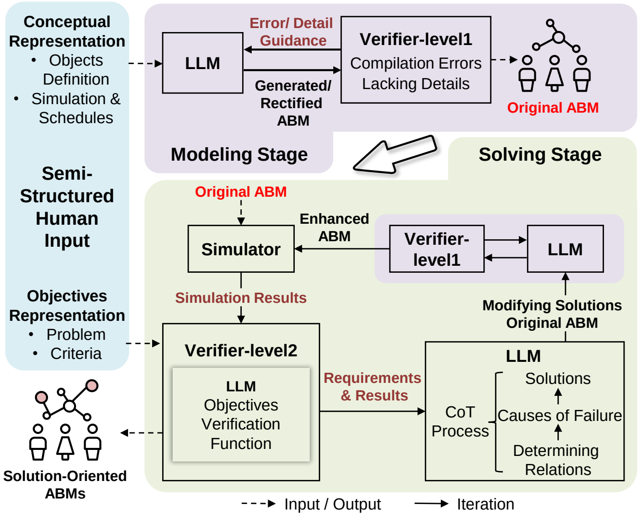
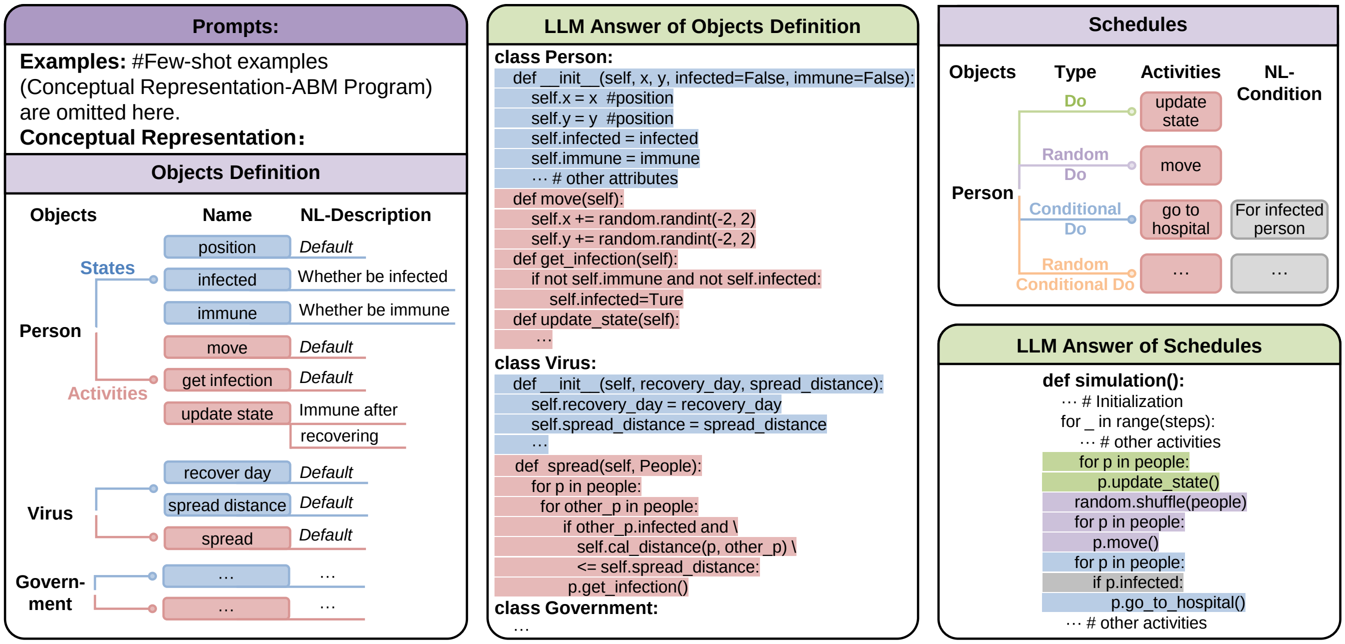


Figure 1: The workflow of SAGE.

3.1 "Modeling" Stage

ABM Conceptual Representation. ABM involves intricate elements such as agents, environments, and their interactions. These elements typically can be instantiated as object states, activities comprising conditions and algorithms for state updating, as well as the scheduling of these activities. However, translating these concepts into practical ABMs requires significant efforts and domain expertise, often proving to be abstract and daunting for non-programmers. To bridge this gap, we design a semi-structured, semi-natural language conceptual representation that serves as a user-friendly guide, allowing users to provide detailed scenario descriptions essential



Examples: #Few-shot examples (Conceptual Representation-ABM Program) are omitted here.
Conceptual Representation:

Objects Definition

Objects		Name	NL-Description
Person	States	position	*Default*
		infected	Whether be infected
		immune	Whether be immune
	Activities	move	*Default*
		get infection	*Default*
		update state	Immune after recovering
Virus		recover day	*Default*
		spread distance	*Default*
		spread	*Default*
Govern-ment		…	…
		…	…

```
class Person:
  def __init__(self, x, y, infected=False, immune=False):
    self.x = x  #position
    self.y = y  #position
    self.infected = infected
    self.immune = immune
    … # other attributes
  def move(self):
    self.x += random.randint(-2, 2)
    self.y += random.randint(-2, 2)
  def get_infection(self):
    if not self.immune and not self.infected:
      self.infected=Ture
  def update_state(self):
    …
class Virus:
  def __init__(self, recovery_day, spread_distance):
    self.recovery_day = recovery_day
    self.spread_distance = spread_distance
    …
  def  spread(self, People):
    for p in people:
      for other_p in people:
        if other_p.infected and \
          self.cal_distance(p, other_p) \
            <= self.spread_distance:
          p.get_infection()
class Government:
  …
```

Objects	Type	Activities	NL-Condition
Person	Do	update state	
	Random Do	move	
	Conditional Do	go to hospital	For infected person
	Random Conditional Do	…	…

```
def simulation():
  … # Initialization
  for _ in range(steps):
    … # other activities
    for p in people:
      p.update_state()
    random.shuffle(people)
    for p in people:
      p.move()
    for p in people:
      if p.infected:
        p.go_to_hospital()
  … # other activities
```

Figure 2: Conceptual representation for ABM description and the generated ABM program.

Algorithm 1: Scenario-specific ABM generation in "Modeling" stage

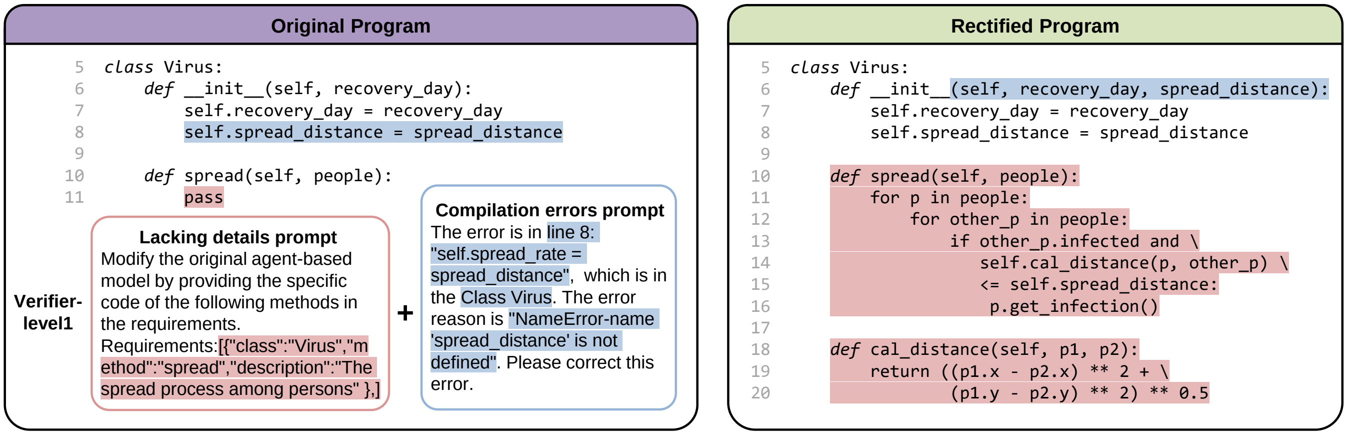
```
Input:Conceptual representation of scenario $\mathcal{L}$, max
  number of iterations $i$ ;
Output:ABM program $A$;
$P_A$ = Prompt_gen_abm($\mathcal{L}$);
$A$ = LLM_generator($P_A$);
defects = Verifier_level1($A$);
while len(defects)>0 and i>0 do
    $P_D$ = Prompt_defects($A$, defects);
    $A$ = LLM_generator($P_D$);
    defects = Verifier_level1($A$);
    $i$ = $i$-1
end
```

for developing ABM models. Meanwhile, leveraging the power of few-shot prompting, LLMs can learn to transform and expand the information encapsulated in the conceptual representation into executable ABM programs.

Figure 2 illustrates a conceptual representation and a generated ABM program. According to the above ABM necessities, the representation consists of two structured components. The first component, "object definition", corresponds to the states and activities of agents and environment objects. The second component, known as "scheduling", corresponds to the execution sequence or trigger conditions of activities. In the object definition, "states" define the descriptive attributes of an object at a specific time step, while "activities" define the dynamic interactions involving the object. For instance, when defining a person in the scenario of epidemic transmission, the states may include "age", "infected", and "immune", while the activities may include "get_infected", "get_immune" and so on. The specific meanings of each state and activity are presented by natural language (NL). Guided by the few-shot prompt (shown in supplementary material), the LLM formalizes the NL descriptions and generalizes the omitted descriptions by utilizing its inherent knowledge and contextual information. In the current implementation, conceptual representations are presented in JSON format.

The simulation process of an ABM involves the flexible scheduling of different agents, each executing their internal functions in a specific order and under certain conditions. To simplify the description of the scheduling of the aforementioned object definition, we abstract and structure them as schedule primitives: *Do(Object_name, Activity_name)* means that all instances of this object perform this activity in the default order; *Random_Do(Object_name, Activity_name)* means that all instances of this object perform this activity in a random order; *Conditional_Do(Object_name, Activity_name, conditions)* means that instances satisfying the conditions of this object perform this activity in the default order; *Random_Conditional_Do(Object_name, Activity_name, conditions)* means that all instances satisfying the conditions of this object perform this activity in a random order. Figure 2 shows an example illustrating the corresponding ABM program with schedule primitives.

Verifier-level1: Program correctness verification. In light of the challenges posed by the lack of self-checking capabilities in LLMs, we devise an automatic ABM program verifier to identify defects in generated programs, ensuring their executability and integrity. These defects are then filled in a prompt template for LLM to further rectify the ABM program, enabling iterative rectification



```
5  class Virus:
6      def __init__(self, recovery_day):
7          self.recovery_day = recovery_day
8          self.spread_distance = spread_distance
9
10     def spread(self, people):
11         pass
```

Lacking details prompt
Modify the original agent-based model by providing the specific code of the following methods in the requirements.
Requirements:[{"class":"Virus","method":"spread","description":"The spread process among persons" },]

Compilation errors prompt
The error is in line 8: "self.spread_rate = spread_distance", which is in the Class Virus. The error reason is "NameError-name 'spread_distance' is not defined". Please correct this error.

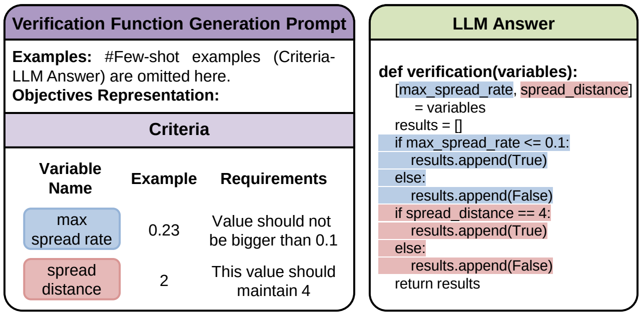
```
5  class Virus:
6      def __init__(self, recovery_day, spread_distance):
7          self.recovery_day = recovery_day
8          self.spread_distance = spread_distance
9
10     def spread(self, people):
11         for p in people:
12             for other_p in people:
13                 if other_p.infected and \
14                    self.cal_distance(p, other_p) \
15                    <= self.spread_distance:
16                     p.get_infection()
17
18     def cal_distance(self, p1, p2):
19         return ((p1.x - p2.x) ** 2 + \
20                 (p1.y - p2.y) ** 2) ** 0.5
```

Figure 3: The rectification prompts generated by verifier-level1 and the rectified ABM program.

until an executable ABM with complete details is obtained. Defects are classified into two types: "compilation errors" and "lacking details". "Compilation errors" hinder proper model execution. For Python programs, they indicate interpretation errors and certain runtime errors. To rectify these errors, verifier-level1 invokes the default compiler or interpreter to compile or run the ABM program, analyzing the returned error messages to assemble the "compilation errors" rectification prompts for LLMs. Each error message is structured in the prompt as $[error\_line, error\_code, error\_reasons]$. On the other hand, defects falling under "lacking details" refer to empty classes or functions that lack substantive implementation details. These defects are often caused by using placeholders instead of detailed code, leaving specific interaction behaviors and updated variable values unaddressed. Verifier-level1 tackles these issues systematically. First, it identifies these defects through text analysis, linking activity names to object classes. This information is then merged with the corresponding activity descriptions from the aforementioned conceptual representation, providing structured guidance for prompt-based rectification. An example of the verifying and rectifying process is shown in Figure 3.

3.2 "Solving" Stage

Objective representation. Under the specific ABM generated by the "Modeling" stage, users may seek solutions or formulate policies addressing distinct problems, such as decreasing the spread rate of a virus in an epidemic model. We design an objective representation, which is also a semi-structured and semi-NL representation, to effectively convey these requirements. The representation comprises two parts: the description of the target problem (e.g. "decrease the spread rate of the virus") and the criteria for judgment. The judgment criteria encompass not only the intended outcomes of the solutions (e.g. "The spread rate of virus is reduced to below 0.1") but also any restrictions (e.g. "The ground truth, like spread distance, should not change"). The target problem is expressed in NL, providing a clear understanding of the specific issue at hand. On the other hand, the criteria typically involve variables that can be quantitatively or qualitatively compared within the described problem scenario. Each criterion is structured as $[variable_{name}, variable_{example}, requirement]$, where $variable_{example}$ specifies the data type of the variable, and $requirement$ described in NL.



Examples: #Few-shot examples (Criteria-LLM Answer) are omitted here.
Objectives Representation:

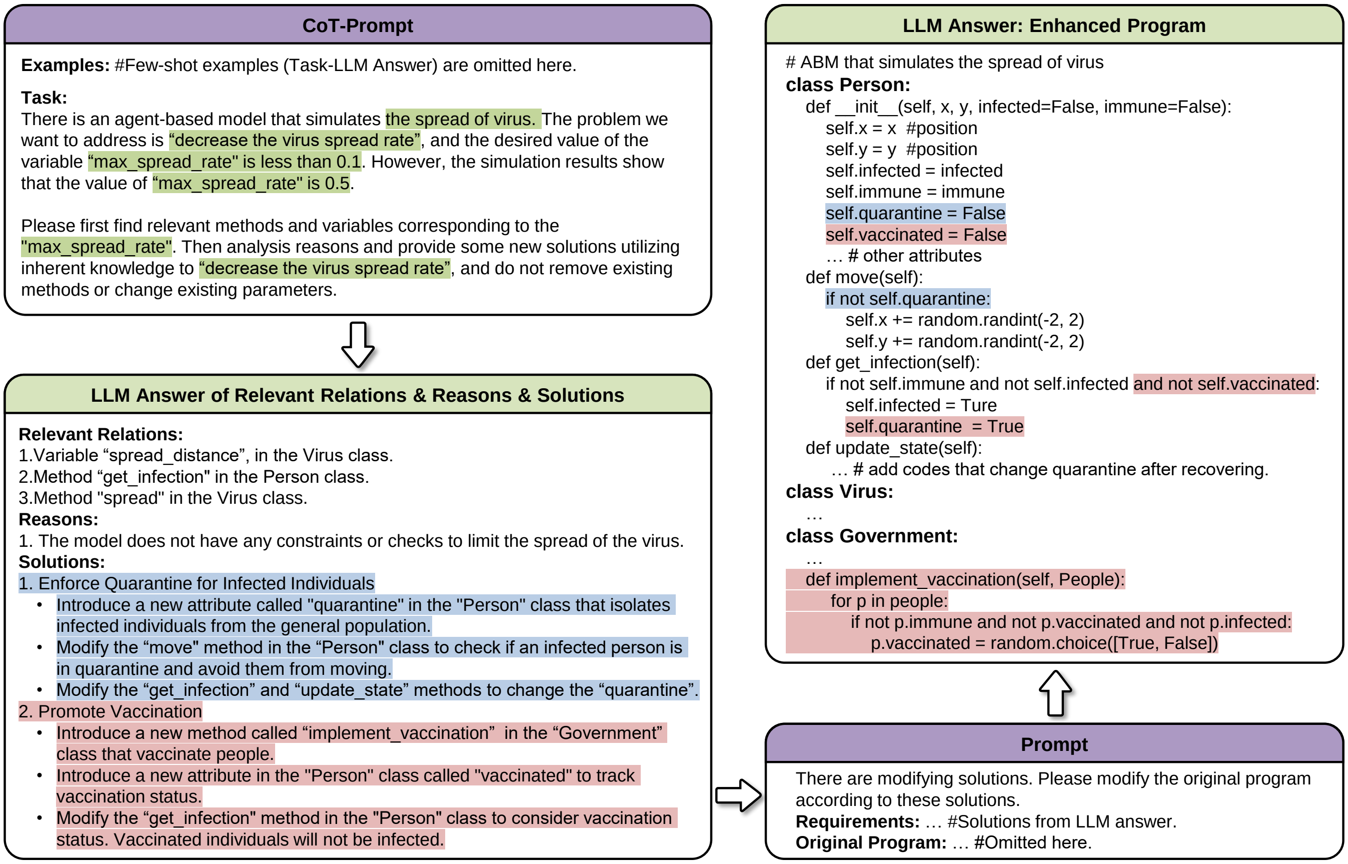
Criteria

Variable Name	Example	Requirements
max spread rate	0.23	Value should not be bigger than 0.1
spread distance	2	This value should maintain 4

```
def verification(variables):
    [max_spread_rate, spread_distance]
        = variables
    results = []
    if max_spread_rate <= 0.1:
        results.append(True)
    else:
        results.append(False)
    if spread_distance == 4:
        results.append(True)
    else:
        results.append(False)
    return results
```

Figure 4: Objective representation for desired solution effects and the generated verification program.

Verifier-level2: Solution effectiveness verification. Verifier-level2 is designed to assess the efficacy of solutions. Using a few-shot prompt, the LLM converts the problem and criteria in the objective representation into verification functions. These functions are crucial as they assess whether the variables extracted from simulation results meet the verification criteria. If all criteria are satisfied, verifier-level2 acknowledges these solutions as successfully verified. Consequently, the executable ABM embedded with these solutions is regarded as the final output of SAGE. Figure 4 showcases a criteria prompt and the corresponding verification function answer for the problem "decrease the spread rate of the virus". Please note that the template of the verification function, including the format of input and output, is provided in the examples within the prompt (omitted in Figure 4). This semi-structured representation combines the flexibility of natural language for describing criteria with the clarity provided by a list structure, allowing for the precise expressions of different criteria. These expressions can be any attribute or complex function (including functions over time series), provided that you record them during simulation and can return the records to the verification function after simulation completion. Moreover, the



CoT-Prompt

Examples: #Few-shot examples (Task-LLM Answer) are omitted here.

Task:
There is an agent-based model that simulates the spread of virus. The problem we want to address is “decrease the virus spread rate”, and the desired value of the variable “max_spread_rate" is less than 0.1. However, the simulation results show that the value of “max_spread_rate" is 0.5.

Please first find relevant methods and variables corresponding to the "max_spread_rate". Then analysis reasons and provide some new solutions utilizing inherent knowledge to “decrease the virus spread rate”, and do not remove existing methods or change existing parameters.

LLM Answer of Relevant Relations & Reasons & Solutions

Relevant Relations:
1.Variable “spread_distance”, in the Virus class.
2.Method “get_infection" in the Person class.
3.Method "spread" in the Virus class.
Reasons:
1. The model does not have any constraints or checks to limit the spread of the virus.
Solutions:
1. Enforce Quarantine for Infected Individuals

- Introduce a new attribute called "quarantine" in the "Person" class that isolates infected individuals from the general population.
- Modify the “move" method in the “Person" class to check if an infected person is in quarantine and avoid them from moving.
- Modify the “get_infection” and “update_state” methods to change the “quarantine”.

2. Promote Vaccination

- Introduce a new method called “implement_vaccination” in the “Government” class that vaccinate people.
- Introduce a new attribute in the "Person" class called "vaccinated" to track vaccination status.
- Modify the “get_infection" method in the "Person" class to consider vaccination status. Vaccinated individuals will not be infected.

Prompt

There are modifying solutions. Please modify the original program according to these solutions.
Requirements: … #Solutions from LLM answer.
Original Program: … #Omitted here.

LLM Answer: Enhanced Program

```
# ABM that simulates the spread of virus
class Person:
  def __init__(self, x, y, infected=False, immune=False):
    self.x = x  #position
    self.y = y  #position
    self.infected = infected
    self.immune = immune
    self.quarantine = False
    self.vaccinated = False
    … # other attributes
  def move(self):
    if not self.quarantine:
      self.x += random.randint(-2, 2)
      self.y += random.randint(-2, 2)
  def get_infection(self):
    if not self.immune and not self.infected and not self.vaccinated:
      self.infected = Ture
      self.quarantine  = True
  def update_state(self):
     … # add codes that change quarantine after recovering.
class Virus:
  …
class Government:
  …
  def implement_vaccination(self, People):
     for p in people:
       if not p.immune and not p.vaccinated and not p.infected:
         p.vaccinated = random.choice([True, False])
```

Figure 5: The CoT workflow and the generated solutions with enhanced ABM program.

verification functions generated by the LLM are written in general-purpose programming languages, enhancing their versatility and comprehensiveness in comparison to combinatorial arrangements of pre-defined criteria implementation.

Solution generation with CoT prompt. Verifier-level2 utilizes the simulation results that fail to meet the evaluation criteria to generate a prompt for the next-stage LLM to enhance the ABM program. Because ABMs involve non-linear dynamics and complex spatiotemporal interactions, it is challenging for LLMs to determine how to modify a model to meet the criteria based solely on a static experimental result. The CoT prompt guides the model through a series of logically connected thinking steps, enabling coherent reasoning and logical thinking. Therefore, we apply the CoT prompt technique to enhance the solution generation of LLMs. Given that the individual state changes in an ABM are fundamentally driven by interactions with the current states of other individuals, our CoT is designed with three steps: relations extraction, which guides LLMs in extracting the operations within the ABM program that influence the evaluated variables; cause analysis, which guides LLMs in analyzing the extracted operations to identify the reasons for unsatisfied criteria; and solution proposal, which guides LLM in leveraging domain knowledge to propose solutions based on the identified causes. Figure 5 illustrates an example on seeking solutions of "decrease spread rate of virus". Following the guidance of the CoT, the LLM first extracts the relevant functions and variables that impact the spread rate, such as variables "spread_distance" and the function "spread". Then, the LLM provides an analysis of the reasons, e.g. lack of constraints or checks contributes to the uncontrolled spread of the virus. Lastly, the LLM proposes two solutions: "enforce quarantine" and "promote vaccination" accompanied by instructions on how to implement them within the original ABM.

4 DATASET AND EVALUATION

4.1 Solution-oriented ABMs Dataset

While several code datasets are available for training or testing LLMs' programming abilities, the majority of them typically consist of isolated code snippets without sufficient contextual information or background knowledge. In particular, there is a lack of evaluation datasets tailored to the generation of ABMs for problem-solving assistance. In this work, we introduce a solution-oriented ABMs dataset aimed at evaluating the utility and effectiveness of the generated ABMs in problem-solving scenarios. This will also promote research and advancements in the fields of natural language processing and programming.

The evaluation dataset is primarily divided into two sub-datasets. The first is the "scenario-model" dataset, which mainly evaluates the language model's ability to generate corresponding ABM programs

Algorithm 2: Scenario-specific solution generation in "Solving" stage

```
Input:objective representation $O$, original executable ABM
  $A$, max number of iterations $i$ ;
Output: ABM program $A$, solutions;
$\mathcal{R}$ = Simulator($A$) # real results ;
$P_V$ = Prompt_gen_Veri($O$);
Veri_func = LLM_generator($P_V$);
satisfying_flag = Verifier_level2(Veri_func, $\mathcal{R}$);
while satisfying_flag==False and i>0 do
    $P_C$ = Prompt_CoT($\mathcal{R}$, $O$, $A$);
    relations, reasons, solutions = LLM_generator($P_C$);
    $P_M$ = Prompt_modification($A$, solutions);
    $A$ = LLM_generator($P_M$);
    $i$ = $i$-1;
    defects = Verifier_level1($A$);
    while len(defects) > 0 do
        $P_D$ = Prompt_defects($A$, defects);
        $A$ = LLM_generator($P_D$);
        defects = Verifier_level1($A$);
    end
    $\mathcal{R}$ = Simulator($A$) # real results ;
    satisfying_flag = Verifier_level2(Veri_func, $\mathcal{R}$);
    if $i$ == 0 then
        break;
    end
end
```

based on the given scenario descriptions. It contains 50 samples, each consisting of a natural language-based scenario description, a conceptual representation-based scenario description as a task, and an executable ABM that programs the scenario from the task description as a ground truth answer. The second part is the "problem-solution" dataset, which primarily assesses the LLMs' capability to propose solutions based on the given ABM for specific problems. This sub-dataset comprises 30 samples, each consisting of a problem description and an executable original ABM as a task, and an executable ABM that comprises the proposed solutions as a ground truth solution answer. In this sub-dataset, some samples contain strict criteria (criteria-defined samples), while others without criteria are open-ended samples. The dataset spans various domains as shown in Table 1 and is primarily sourced from open-source projects on GitHub. Descriptions of the scenarios and problems are derived from project introductions and comments within the code, which are subsequently supplemented and standardized through manual efforts. The code segments in the dataset are extracted from the code of corresponding projects, meticulously organized and refined.

4.2 "Modeling" Evaluation

To evaluate the practical applicability of SAGE in modeling, we conduct experiments using the "scenario-model" sub-dataset. Four state-of-the-art LLMs: GPT-3.5-Turbo, Code-Llama-34b, Claude-instant, and GPT-4 are chosen as the generative models. These LLMs generate ABMs according to different scenario descriptions from the dataset under two conditions: one involving the generation process with SAGE, and the other without SAGE (No-SAGE). Specifically, no-SAGE process refers to LLMs generating ABMs and solutions solely based on natural language descriptions without using SAGE's semi-structured representation or two-level verification optimization. The generated ABMs are then compared against their corresponding ground truth models to evaluate the generation quality. To ensure fairness, given SAGE's iterative nature in the generation process, each sample undergoes 10 times of modeling in the No-SAGE condition. The best of these results is then used for subsequent comparisons. The "10" is same with the maximum number of iterations set in the SAGE-involved experiments.

Table 1: Dataset description

	Domains
scenario-model dataset	Public health: 22%; Society: 22%; Ecology: 12%; Traffic: 10%; Economics: 10%; Biology: 10%; Education: 8%; Gaming: 6%;
problem-solution dataset	Criteria-defined samples: 40% Open-ended samples: 60%

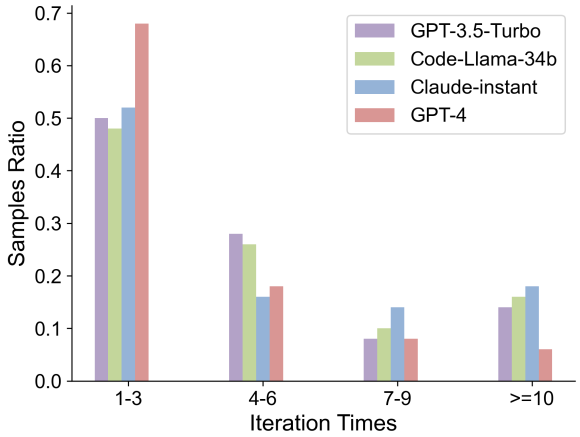


Figure 6: Iterations times for ideal ABMs generation

Evaluation metrics Two metrics are employed to evaluate the quality of the generated ABMs. The first metric involves measuring the similarity between the generated model and the ground truth answer using CodeBLEU[2], which weights the average of the lexical, abstract syntax tree (AST), and data flow match between the generated code and the ground truth. We reduce the weight given to lexical match and improve the weight given to AST and dataflow match during the evaluation, since the sequence of object definitions does not impact the ultimate simulation outcomes. The second metric evaluates whether the generated ABMs are executable and elaborate, which is conducted by the verifier-level1 of SAGE.

Evaluation results The first two columns of Table 2 indicate that ABMs produced using SAGE with LLMs consistently yield

Table 2: Evaluation results on executable ABMs generation

LLMs	CodeBLEU No-SAGE	CodeBLEU SAGE	Executable No-SAGE	Executable SAGE	Elaborate No-SAGE	Elaborate SAGE
GPT-3.5-Turbo	42.47	52.18	64.00	90.00	68.00	90.00
Code-Llama-34b	44.94	53.36	64.00	86.00	78.00	88.00
Claude-instant	42.49	49.64	48.00	82.00	74.00	84.00
GPT-4	45.73	53.29	78.00	96.00	74.00	96.00

higher scores (four LLMs' average improvement of 18.7%) compared to those generated by No-SAGE, when applying CodeBLEU for assessment.From the perspective of the executability and integrity of the generated ABMs, the last four columns of Table 2 present the ratios of the generated ABMs that exhibit these qualities among all answers. These data underscore SAGE's remarkable efficacy in facilitating error correction and content enrichment of ABMs, as evident across all four LLMs. Figure 6 illustrates the number of iterations required for different LLMs to generate an executable and elaborate ABM when utilizing SAGE (Generation failed samples are also included in the category of 10 or more iterations). The results show that with the incorporation of SAGE, over 70% of the samples can be resolved within six iterations or fewer.

4.3 "Solving" Evaluation

In the "Solving" evaluation experiments, we employ the "problem-solution" dataset as the testing set and utilize two general-purpose LLMs, GPT-3.5-Turbo and GPT-4, as the solution generation models. The impact of SAGE on the successful resolution of the problem is assessed by comparing the success rates of problem resolution between the scenarios with and without the use of SAGE. Similar to the "modeling" evaluation experiments, each sample undergoes 5 times of "solving process" in the No-SAGE condition, and the best solution-oriented ABM is chosen. Besides, to assess the relative importance of different verification levels in the "solving" process, we analyze the failure reasons in all unsuccessful cases.

Evaluation metrics In addition to assessing the executability of the generated solutions, for samples with pre-defined criteria, we determine the success of problem resolution based on whether the simulation results meet the evaluation criteria. For samples without specific criteria, which are named as open-ended problems, we manually evaluate solutions' practical significance, which primarily consist of two standards. Firstly, from the perspective of object definitions, we examine whether there are meaningful additions or deletions of states and activities of objects, rather than simple modifications to the parameters involved in the existing ABMs. Secondly, from the perspective of scheduling, we assess whether the added or deleted states or activities truly affect the simulation process, in other words, whether they are invoked during the execution process. The solutions of one sample is confirmed only when all of these standards are simultaneously fulfilled.

Evaluation results The evaluation results (Table 3) indicate the problem-solving rates. It is evident that when comparing problem solving rates with and without SAGE involvement, there is a noticeable enhancement in problem-solving efficiency under the SAGE involvement for both criteria-defined samples and open-ended samples (average improvement of 38.1%). The verifier-level1 proves effective in preventing execution errors, while the verifier-level2 facilitates the generated solutions meet the pre-defined criteria and content standards. Figure 7 shows that both types of above defects cannot be overlooked. Therefore, in order to guarantee the generation of executable and effective solutions, both levels of verifiers are indispensable.

Table 3: Problem solving-rates on Criteria-defined(CD) samples and Open-ended(OE) samples

LLMs	CD No-SAGE	CD SAGE	OE No-SAGE	OE SAGE
GPT-3.5-Turbo	58.33	91.67	38.89	88.89
GPT-4	41.67	100	66.67	77.78

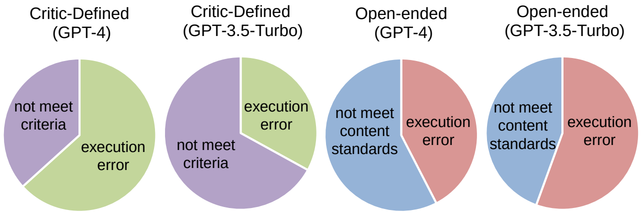


Figure 7: Causes statistics for unsuccessful solutions

5 CONCLUSIONS

In this paper, we propose a solution-oriented ABM generation framework: SAGE. SAGE utilizes the powerful abilities of LLMs, including in-context learning, natural language understanding, compositional generalization, and domain-knowledge retrieval. This enables the development of a two-stage, iterative procedure for automatic agent-based modeling and solution generation. Our results demonstrate a significant improvement in generation performance compared to traditional No-SAGE approaches. Notably, SAGE achieves this without requiring the extensive training process of LLM and a large amount of data. Thus, SAGE has the ability to significantly reduce the barriers to creating ABMs and utilizing them for problem-solving, enhancing the productivity of complex system research.

ACKNOWLEDGMENTS

This work was partly supported by the National Key Research and Development Program of China (grant no. 2021ZD0200300), National Nature Science Foundation of China (nos. 62088102 and 61836004), IDG/McGovern Institute for Brain Research at Tsinghua University

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[3] Xavier Daull, Patrice Bellot, Emmanuel Bruno, Vincent Martin, and Elisabeth Murisasco. 2023. Complex QA and language models hybrid architectures, Survey. *arXiv preprint arXiv:2302.09051* (2023).

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[5] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey for in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[7] Tatiana Filatova, Peter H Verburg, Dawn Cassandra Parker, and Carol Ann Stannard. 2013. Spatial agent-based models for socio-ecological systems: Challenges and prospects. *Environmental modelling & software* 45 (2013), 1–7.

[8] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.

[9] G.Inc. 2023. Copilot. https://github.com/features/copilot

[10] Rory Greig and Jordi Arranz. 2021. Generating agent based models from scratch with genetic programming. In *ALIFE 2021: The 2021 Conference on Artificial Life*. MIT Press.

[11] Volker Grimm, Gary Polhill, and Julia Touza. 2017. Documenting social simulation models: the ODD protocol as a standard. *Simulating social complexity: A handbook* (2017), 349–365.

[12] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[13] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[14] Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14953–14962.

[15] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[16] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and individual differences* 103 (2023), 102274.

[17] Franziska Klügl and Ana LC Bazzan. 2012. Agent-based modeling and simulation. *Ai Magazine* 33, 3 (2012), 29–29.

[18] Raphael Koster, Jan Balaguer, Andrea Tacchetti, Ari Weinstein, Tina Zhu, Oliver Hauser, Duncan Williams, Lucy Campbell-Gillingham, Phoebe Thacker, Matthew Botvinick, et al. 2022. Human-centred mechanism design with Democratic AI. *Nature Human Behaviour* 6, 10 (2022), 1398–1407.

[19] William B Langdon and Riccardo Poli. 2013. *Foundations of genetic programming*. Springer Science & Business Media.

[20] David Lazer, Alex Pentland, Lada Adamic, Sinan Aral, Albert-László Barabási, Devon Brewer, Nicholas Christakis, Noshir Contractor, James Fowler, Myron Gutmann, et al. 2009. Computational social science. *Science* 323, 5915 (2009), 721–723.

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[22] Charles Edward Lindblom, David K Cohen, et al. 1979. *Usable knowledge: Social science and social problem solving*. Vol. 21. Yale University Press.

[23] Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. 2022. Commitbart: A large pre-trained model for github commits. *arXiv preprint arXiv:2208.08100* (2022).

[24] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. *arXiv preprint arXiv:2301.09072* (2023).

[25] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[26] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786* (2021).

[27] Michael Macy and Andreas Flache. 2011. Social dynamics from the bottom up: Agent-based models of social interaction. (2011).

[28] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651* (2023).

[29] David Masad, Jacqueline Kazil, et al. 2015. MESA: an agent-based modeling framework. In *14th PYTHON in Science Conference*, Vol. 2015. Citeseer, 53–60.

[30] Bridgette Parsons, José M Vidal, Nathan Huynh, and Rita Snyder. 2015. Automatic generation of agent behavior models from raw observational data. In *Multi-Agent-Based Simulation XV: International Workshop, MABS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*. 121–132.

[31] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).

[32] Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. 2022. Reasoning with language model prompting: A survey. *arXiv preprint arXiv:2212.09597* (2022).

[33] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[34] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[35] Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. 2023. Errors are Useful Prompts: Instruction Guided Task Programming with Verifier-Assisted Iterative Prompting. *arXiv preprint arXiv:2303.14100* (2023).

[36] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature medicine* (2023), 1–11.

[37] Seth Tisue and Uri Wilensky. 2004. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, Vol. 21. Citeseer, 16–21.

[38] Arthur Turrell. 2016. Agent-based models: understanding the economy from the bottom up. *Bank of England Quarterly Bulletin* (2016), Q4.

[39] Boshi Wang, Xiang Deng, and Huan Sun. 2022. Iteratively prompt pre-trained language models for chain of thought. *arXiv preprint arXiv:2203.08383* (2022).

[40] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481* (2022).

[41] Duncan J Watts. 2017. Should social science be more solution-oriented? *Nature Human Behaviour* 1, 1 (2017), 0015.

[42] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[43] Themis Dimitra Xanthopoulou, Andreas Prinz, and F LeRon Shults. 2019. Generating executable code from high-level social or socio-ecological model descriptions. In *System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0: 11th International Conference, SAM 2019, Munich, Germany, September 16–17, 2019, Proceedings 11*. 150–162.

[44] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (2022).

[45] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*. PMLR, 12697–12706.

[46] Stephan Zheng, Alexander Trott, Sunil Srinivasa, David C Parkes, and Richard Socher. 2022. The AI Economist: Taxation policy design via two-level deep multiagent reinforcement learning. *Science advances* 8, 18 (2022), eabk2607.

[47] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).

[48] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).