Co-Synthesis of Code and Formal Models Using Large Language Models and Functors

Sumit Kumar Jha*, Susmit Jha†, Rickard Ewetz‡, Alvaro Velasquez§
*Computer Science Department, Florida International University, Miami, FL, USA, jha@cs.fiu.edu
†Computer Science Laboratory, SRI International, Menlo Park, CA, USA, susmit.jha@sri.com
‡ECE Department, University of Florida, Gainesville, FL, USA, rickard.ewetz@ufl.edu
§Department of Computer Science, University of Colorado Boulder, CO, USA, alvaro.velasquez@colorado.edu

***Abstract*—Large Language Models (LLMs) have demonstrated remarkable capabilities in generating code from natural language descriptions, including code for parallel systems. However, ensuring that this code is free of errors, particularly in concurrent and synchronization-heavy contexts, remains a significant challenge. In this paper, we introduce a new approach that employs LLMs to co-synthesize the code, its formal model and the functor that establishes the mapping between the code and its formal model. The formal models are then verified against temporal logic specifications using model checking. The functors serve as human-auditable artifacts that help establish equivalence between the code and its formal model. Our methodology is demonstrated through experiments involving the co-synthesis of C code and its formal model for the dining philosophers problem. Our experimental results using models from OpenAI, Anthropic, and Meta evaluate the effectiveness of our approach.**

I. Introduction

The rapid advancements in Large Language Models (LLMs) have unlocked new possibilities for generating code directly from natural language descriptions. These models are increasingly capable of understanding complex instructions and translating them into executable code across various programming languages. However, the task of ensuring that the generated code meets the stringent requirements of concurrency and synchronization correctness remains a challenging problem, particularly in parallel systems. These errors can be subtle but catastrophic in their consequences, making the verification of parallel code a critical aspect of high-performance software development.

In this paper, we present a new approach that seeks to address this challenge by not only generating code but also co-synthesizing a formal model and a corresponding functor. This functor establishes a human-auditable mapping between the code and its formal verification model, providing a robust framework for high-assurance systems [1]. The formal models are designed to capture the essential behaviors and properties of the generated code, such as synchronization, which are then verified against temporal logic specifications using model checking. By leveraging functors, rooted in category theory, we bridge the gap between the formal models and the concrete implementation, ensuring that the generated code and its formal model are more likely to be consistent with each other.

TABLE I: Performance of Different Models on Co-Synthesis of Code, Formal Models, and Associated Functors

Source	AI Model	C Code #Iterations	SMV Model #Iterations	Functor Quality	Model Checking
OpenAI	GPT-4o	2	4	✓	✓
Anthropic	Claude-3.5 Sonnet	1	1	✓	✓
Meta	Llama-3.1 405B	15	14	✗	✓

Our approach can potentially reduce the manual effort required in verifying LLM-generated code. By automating the generation of both the code and its formal model, and by establishing a human-auditable mapping between the code and its models using functors, our method streamlines the formal verification process, seeking to make formal analysis more popular by building formal models more efficiently.

Our methodology is demonstrated through a series of experiments, particularly focusing on the dining philosophers problem – a classic in synchronization. The results of our experiments, as shown in Table I, show that our approach successfully generates code and formal models which are then verified against temporal logic properties. In this paper, we make the following contributions:

(i) We introduce a new co-synthesis approach that generates (i) code, (ii) formal models, and (iii) functor mapping between the code and its model using LLMs.
(ii) We highlight that the functor between the code and its formal model enhances human trust in the mapping between the formal model and the code.
(iii) We successfully apply the approach to the dining philosophers problem, demonstrating the generation of code, verified models and functors using OpenAI's GPT-4o and Anthropic's Claude-3.5 Sonnet models.
(iv) We successfully verify the generated models against temporal logic specifications for 3 to 27 processes using symbolic model checking.

Our approach risks enhancing human trust in flawed code, as the functor may inadvertently lead to a human relating the code with an incorrect model, leading to misplaced trust in code that does not align with the verified model.

II. Related Work

A. Category Theory

Category theory has been widely recognized as a powerful mathematical framework for modeling abstract structures and their relationships [2]. More recently, category theory has been applied in sciences, data science and other computational domains [3]. The use of functors, in particular, has been studied to formalize the relationship between different computational structures for computer scientists and logicians [4], [5]. In this paper, we employ functors to relate formal models to code in a human-auditable manner.

B. Large Language Models

Large Language Models (LLMs) like GPT-4 have shown significant capabilities [6], including the ability to generate complex code from natural language prompts. The Codex model, for example, has been used extensively to generate code across various programming languages [7]. However, the limitations of LLMs in creating correct code have been noted in literature [8]. Moreover, application of these models to more intricate tasks such as formal verification of planning problems has also been explored [5], [9], [10]. While there has been work in high-assurance AI [1], [11], [12], our focus in this paper is to co-synthesize code, models and functors.

C. Code Synthesis

Program synthesis has seen significant advancements with the integration of machine learning techniques. An extensive survey on program synthesis [13] focuses on inductive synthesis from examples. The use of counterexample-guided inductive synthesis (CEGIS) for generating correct by construction programs has led to interesting results [14], [15]. Additionally, sketching-based synthesis approaches have also contributed to the automated synthesis of programs [16]. Based on these foundational results, a number of practical systems have been built to assist programmers with the aid of large language models. In this paper, we have used state-of-the-art large language models that have been trained on code, text and other data to enable the co-synthesis of not just code but also formal models and functors associating the code with the formal model.

D. Model Checking and Temporal Logic

Model checking is a cornerstone in the verification of concurrent systems. Clarke, Emerson, and Sifakis were pioneers in this field, for which they received the Turing Award [17]. Model checking [18], [19] has benefitted from the introduction of temporal logics, particularly linear temporal logic (LTL) and computation tree logic (CTL), into computer science [20]. In this paper, we employ model checking to formally verify the correctness of the automatically synthesized formal models against correctness properties stated in temporal logic. Lightweight verification techniques, such as statistical model checking [21], may also be used to detect bugs in concurrent systems.

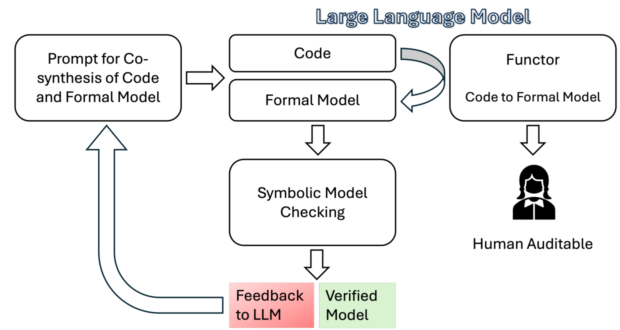

Fig. 1: Overview of the approach to co-synthesize code, formal models and functors for code synthesis using LLMs. A prompt is fed to an LLM that generated the code, the formal model and functor mapping the code to the model. The formal model is verified against temporal logic properties for correctness using model checking. Counterexamples are fed back into the LLM to improve the quality of the code, the model and the functor. The functor can be studied by a human expert to confirm that the formal model is indeed a reliable approximation of the source code.

III. Technical Approach

Our approach leverages Large Language Models (LLMs) to co-synthesize code, the corresponding formal model, and the functor that establishes a human-auditable mapping between the two. This co-synthesis process allows human beings to more readily check that the generated code agrees with the formal model by examining the functor mapping between the code and the model. The formal models are verified against temporal logic specifications using model checking, which provides assurance guarantees for critical properties, including those describing safety and liveness guarantees.

A. Overview

Our co-synthesis approach begins with an LLM tasked with generating executable code for a specific problem, such as the dining philosophers problem. Alongside the executable code, the LLM also generates a formal model, such as a description in the Symbolic Model Verifier (SMV) language and a functor that maps the elements of the executable code to their corresponding elements in the formal model. The functor serves as a bridge, ensuring that a human being can more readily verify that the formal model accurately reflects the behavior of the code, thereby enabling effective verification through model checking. As depicted in Fig. 1, co-synthesis approach is iterative and goes through the following steps:

1) Code Generation: The LLM generates the executable code for the given problem. In our study, we have generated C code focusing on synchronization and concurrency control mechanisms as these serve as good benchmarks for detecting bugs using model checking.
2) Formal Model Generation: Concurrently, the LLM synthesizes a formal model that represents the same logic

as the generated C code. In our work, we have used the NuSMV model checker and employed the SMV language to represent the formal model being synthesized by the large language model.

3) Functor Mapping: As the LLM generated the code and the formal model, it concurrently also formulates a functor that maps objects and morphisms in the C code to their counterparts in the SMV model. This functor is crucial for ensuring the equivalence between the code and the model. Currently, we use human auditing to ensure that the functor clearly establishes a mapping between the code and the formal model; however, this could potentially be automated in the future.
4) Model Checking: The formal model is verified using model checking against a set of temporal logic specifications. If the model checking ensures that the models is correct, our trust in the correctness of the generated code is enhanced due to the presence of the human-auditable functor. If the model checking obtains a bug, the counterexample is fed back to the LLM asking for new variants of the code, the formal model, and the associated functor mapping the code to the model.

B. Categories and Functors

In this context, categories and functors provide the mathematical foundation for mapping the elements of the generated code to the formal model. The functor seeks to ensure that the structure and behavior of the code are preserved when translated into the formal model, and that this mapping can be more readily audited by human experts.

Definition III.1. A category $\mathcal{C}$ consists of:

(i) A class of objects $\mathrm{Ob}(\mathcal{C})$.
(ii) For each pair of objects $A, B \in \mathrm{Ob}(\mathcal{C})$, a set of morphisms $\mathrm{Hom}_{\mathcal{C}}(A, B)$ from A to B.
(iii) A binary operation called composition, defined on compatible pairs of morphisms. For any morphisms $u : A \to B$ and $v : B \to C$, there exists a morphism $v \circ u : A \to C$.

In the case of programs and models, the state of the program or model serve as a class of objects. An instruction serves as a morphism, mapping one state of the program or model to another. Composition naturally comes to programs or models as described by sequence of instructions executing one after another.

Definition III.2. A functor F between two categories $\mathcal{C}$ and $\mathcal{D}$ is a structure-preserving mapping, consisting of:

(i) An assignment to each object $A \in \mathrm{Ob}(\mathcal{C})$, an object $F(A) \in \mathrm{Ob}(\mathcal{D})$.
(ii) An assignment to each morphism $u \in \mathrm{Hom}_{\mathcal{C}}(A, B)$, a morphism $F(u) \in \mathrm{Hom}_{\mathcal{D}}(F(A), F(B))$, such that:
   (a) Identity Preservation: For every object $A \in \mathrm{Ob}(\mathcal{C})$, $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$.
   (b) Composition Preservation: For all morphisms $u : A \to B$ and $v : B \to C$ in $\mathcal{C}$, $F(v \circ u) = F(v) \circ F(u)$.

In our approach, the functor F is responsible for mapping the objects and operations from the category representing the C code to the category representing the formal model in SMV. The functor preserves the logical structure of the code, ensuring that each operation in the code has a corresponding operation in the formal model.

In case of code and models, the identity preservation is achieved using the no-op operation. Composition in this context is achieved as mapping a state from the code to the model afer a sequence of operations is indeed the same as performing the corresponding operations directly on the formal model.

IV. Experiments

To illustrate our approach, we applied it to the classic dining philosophers problem. The LLM was tasked with generating C code to manage synchronization between the dining philosophers. The LLM was also asked to produce a corresponding formal model in SMV and a functor that maps the elements of the C code to the formal model.

We evaluated our methodology using models from OpenAI [22], Anthropic, and Meta [23], evaluating the efficacy of different LLMs. Our experiments focus on the co-synthesis of C code and formal models for the dining philosophers problem. Our co-synthesis approach involves the following steps. LLMs are used to generate C code for synchronization constructs such as the dining philosophers problem. Simultaneously, LLMs also generate equivalent formal models and functors that map the generated code to formal models in the SMV language. The formal models are verified against temporal logic specifications using model checking. If the model checker identifies a violation of the specified temporal logic properties, the LLM uses this feedback to refine the code, the formal model, or both. This iterative refinement ensures that the formal model corresponding to the final code is verifiable through model checking, and the functors enable us to raise our confidence in the generated code.

We used the following prompt in our experiments:

> Write tightly coupled SMV and C code for the Dining Philosophers problem with N philosophers and N forks so that a functor between the two is clearly defined in a syntactic manner, allowing objects and morphisms to be mapped from C to SMV code in a well-structured way. Include a test harness for the C code with assertions to detect synchronization violations and ensure proper resource allocation. The SMV code should include temporal logic specifications to enforce the following conditions:
>
> - No two adjacent philosophers can eat simultaneously, i.e., two philosophers sharing a fork cannot be in the eating state at the same time.
> - Each philosopher eventually gets to eat, ensuring no philosopher starves.
> - A philosopher must pick up both forks to his left and his right before starting to eat.
>
> Define the functor explicitly by mapping the objects (e.g., forks) and morphisms (e.g., actions like picking up and putting down forks) from the C code to the SMV code, ensuring syntactic clarity and consistency. Specify the category for the C code and the SMV code.
>
> Here is an unrelated simple example to illustrate the SMV language constructs. Do not create new SMV constructs on your own. It may be simple to define each philosopher and fork separately.

The prompt was followed by a one-shot examples of an unrelated model the SMV language.

A. GPT-4o Results

We applied our approach to the classic dining philosophers problem. The LLM generated C++ code that passed assertion checks but the corresponding first SMV formal model generated by GPT had syntactic errors and then two subsequent generated formal models failed the temporal logic specifications. At the fourth iteration, a model was produced that passed the temporal logic specifications; see model fragment below.

```
MODULE main
VAR
    fork_0 : boolean; fork_1 : boolean; fork_2 : boolean;
    philosopher_0 : process Philosopher(0, fork_0, fork_1);
    philosopher_1 : process Philosopher(1, fork_1, fork_2);
    philosopher_2 : process Philosopher(2, fork_2, fork_0);
ASSIGN
    init(fork_0) := TRUE; init(fork_1) := TRUE; init(fork_2) := TRUE;
MODULE Philosopher(id, left_fork, right_fork)
VAR
    state : {thinking, hungry, eating};
ASSIGN
    init(state) := thinking;
    next(state) := case
        state = thinking : {thinking, hungry};
        state = hungry & left_fork & right_fork : eating;
        state = eating : thinking;
        TRUE : state;
    esac;
    next(left_fork) := case
        state = hungry & left_fork & right_fork : FALSE;  -- Occupy the
            left fork
        state = eating : TRUE;  -- Release the left fork
        TRUE : left_fork;
    esac;
    next(right_fork) := case
        state = hungry & left_fork & right_fork : FALSE;  -- Occupy the
            right fork
        state = eating : TRUE;  -- Release the right fork
        TRUE : right_fork;
    esac;
```

The corresponding C code generated by the LLM corresponds to the following code fragment:

```
pthread_mutex_t forks[N];
pthread_t philosophers[N];
int states[N]; // 0: thinking, 1: hungry, 2: eating

void pickup_forks(int phil) {
    int left = phil; int right = (phil + 1) % N;
    // Pick up forks (must do in a specific order to prevent deadlock)
    if (phil % 2 == 0) {
        pthread_mutex_lock(&forks[left]);  // Pick up left fork
        pthread_mutex_lock(&forks[right]); // Pick up right fork
    } else {
        pthread_mutex_lock(&forks[right]); // Pick up right fork
        pthread_mutex_lock(&forks[left]);  // Pick up left fork
    }
    states[phil] = 2; // Eating
}

void putdown_forks(int phil) {
    int left = phil; int right = (phil + 1) % N;
    states[phil] = 0; // Thinking
    pthread_mutex_unlock(&forks[left]);  // Put down left fork
    pthread_mutex_unlock(&forks[right]); // Put down right fork
}

void* philosopher(void* num) {
    int phil = *(int*)num;
    while (1) {
        states[phil] = 0; // Thinking
        states[phil] = 1; //Hungry
        pickup_forks(phil); // Eating
        putdown_forks(phil);
    }
}
```

As shown in Table II, the LLM also generates a functor mapping between the C code and the SMV model that can enable a user to gain confidence in the correctness of the C code once the formal model has been formally verified.

Temporal Logic Model Checking: We defined three temporal logic specifications in SMV to ensure the correctness and proper synchronization of the Dining Philosophers model.

TABLE II: Functor generated by the GPT model between the C code and the SMV Model

C Object/Morphism	SMV Object/Morphism	Description
`philosopher[i]`	`philosopher_i`	Maps each philosopher thread to an SMV process.
`fork[i]`	`fork_i`	Maps each fork to a boolean variable in SMV.
`pickup_forks(i)`	`state = hungry & left_fork & right_fork : eating`	Maps the action of picking up forks to a state transition.
`putdown_forks(i)`	`state = eating : thinking`	Maps the action of putting down forks to a state transition.
`states[i]`	`state`	Maps the philosopher's state to the corresponding SMV state.

1) No two adjacent philosophers can eat simultaneously: This specification ensures that no two adjacent philosophers, who share a fork, can be in the 'eating' state at the same time.

```
SPEC AG !(philosopher_0.state = eating & philosopher_1.
      state = eating);
SPEC AG !(philosopher_1.state = eating & philosopher_2.
      state = eating);
SPEC AG !(philosopher_2.state = eating & philosopher_0.
      state = eating);
```

These specifications assert that in every reachable state, it is never the case that two adjacent philosophers are simultaneously in the 'eating' state.

2) Each philosopher eventually gets to eat: This specification guarantees that each philosopher who is in the 'hungry' state will eventually transition to the 'eating' state.

```
SPEC AG (philosopher_0.state = hungry -> AF philosopher_0
      .state = eating);
```

3) A philosopher must pick up both forks before eating: This specification ensures that a philosopher can only be in the 'eating' state if both the forks to their left and right are not available, i.e., both forks are occupied by the philosopher.

```
SPEC AG (philosopher_0.state = eating -> !fork_0 & !
      fork_1);
```

TABLE III: Verification Results for the Formal Model

#Philosophers	Memory (MB)	Time (sec)
3	15.636	0.034
4	16.284	0.117
5	16.680	0.041
10	20.512	0.207
20	68.904	11.613
25	82.524	33.637
27	98.936	997.44
30	1078.508	Timeout (8 hours)

The results of our experiments are summarized in Table III, which highlights the performance of our approach in terms of memory usage and verification time across different configurations of the dining philosophers problem.

B. Claude-3.5 Sonnet Results

We present the results obtained using the Claude 3.5 language model for the co-synthesis of C code, SMV formal model, and the functor associating the two for the Dining Philosophers problem.

Claude was able to generate both the C code, the SMV model, and the associated functor in a single iteration without requiring any feedback. The SMV formal model satisfied all the specified temporal logic properties. A fragment of the corresponding SMV formal model generated by Claude 3.5 is shown below:

```
MODULE main
VAR
  philosopher0 : process philosopher(0, semaphore0, semaphore2);
  philosopher1 : process philosopher(1, semaphore1, semaphore0);
  philosopher2 : process philosopher(2, semaphore2, semaphore1);
  semaphore0 : boolean; semaphore1 : boolean; semaphore2 : boolean;

ASSIGN
  init(semaphore0) := FALSE; init(semaphore1) := FALSE; init(semaphore2)
        := FALSE;

MODULE philosopher(id, left_fork, right_fork)
VAR
  state : {thinking, hungry, eating};

ASSIGN
  init(state) := thinking;
  next(state) :=
    case
      state = thinking : {thinking, hungry};
      state = hungry & !left_fork & !right_fork : eating;
      state = eating : {eating, thinking};
      TRUE : state;
    esac;

  next(left_fork) :=
    case
      state = hungry & !left_fork & !right_fork : TRUE;
      state = eating & next(state) = thinking : FALSE;
      TRUE : left_fork;
    esac;

  next(right_fork) :=
    case
      state = hungry & !left_fork & !right_fork : TRUE;
      state = eating & next(state) = thinking : FALSE;
      TRUE : right_fork;
    esac;
```

As shown in Table IV, the Claude 3.5 model successfully generated a functor mapping between the C code and the SMV model. The functor provides a syntactic mapping between the C and SMV implementations, which allows for structured human verification of the Dining Philosophers problem between the two representations.

TABLE IV: Claude-3.5 Sonnet Generated Functor Mapping

C Object / Morphism	SMV Object / Morphism	Description
`philosopher[i]`	`philosopher_i`	Maps each philosopher thread to an SMV process.
`fork[i]`	`semaphore_i`	Maps each fork to a boolean variable in SMV.
`pickup_forks(i)`	`state = hungry & left_fork & right_fork : eating`	Maps the action of picking up forks to a state transition.
`putdown_forks(i)`	`state = eating : thinking`	Maps putting down forks action to a state transition.
`states[i]`	`state`	Maps the philosopher's state to the SMV state.

C. Llama-3.1 405B Results

Llama 3.1 405B was also tasked with generating both C code and SMV formal models coupled with a functor mapping the two representations for the classic Dining Philosophers problem. Several issues were identified during the experiment:

1) Incorrect state management: The code struggled with managing the concurrent states of philosophers, leading to scenarios where multiple philosophers could enter the 'eating' state simultaneously.
2) Compilation errors: The initial C code generated by Llama 3.1 contained compilation errors, including the absence of a state field in the fork structure, which was incorrectly referenced in the code. This required explicit feedback to the LLM model.

Despite these challenges, with significant manual intervention, a working version of the C code was produced.

The SMV model generated by Llama 3.1 aimed to represent the behavior of the philosophers and forks as a finite state machine. However, several challenges were observed:

1) Syntax errors: The initial SMV model contained several syntax errors, including incorrect array declarations and the use of unsupported operators. These errors had to be corrected to make the model syntactically valid.
2) Failed specifications: The model failed to satisfy a few of the key specifications. For example, a counterexample generated by NuSMV demonstrated that two adjacent philosophers could indeed eat simultaneously, which violated the mutual exclusion property. Counterexamples were provided as feedback to Llama to correct such behaviors.
3) Manual feedback for corrections: Significant manual intervention was required to correct both the C code and the SMV model. This included missing synchronization mechanisms, syntax errors, and adjusting the logic to better align with the requirements.

Our experiments with Llama 3.1 showed that the model did not perform as well as the earlier models.

TABLE V: Llama 3.1 405B Generated Functor Mapping

C Object/Operation	SMV Object/Operation	Description
`philosopher[i]`	`philosopher_i`	Maps each philosopher thread to an SMV process.
`fork[i]`	`fork_i`	Maps each fork to a boolean variable in SMV.
`pickup_fork(i)`	`next(fork_i)`	Maps the action of picking up forks to a state transition.
`putdown_fork(i)`	`next(fork_i)`	Maps the action of putting down forks to a state transition.
`eat(i)`	`next(philosopher_i)`	Maps the action of eating to a state transition in SMV.

As shown in Table V, the functor mapping between the C code and the SMV model was also challenging. The objects, such as philosophers and forks, could be mapped conceptually, but the morphisms (functions and transitions) do not instill confidence in a human reviewer that the C code and the formal model are correctly aligned. For example, both pickup_fork(i) and putdown_fork(i) are mapped to

next(fork_i). Overall, Llama 3.1 struggled to generate correct and consistent C code and SMV models for the Dining Philosophers problem. The experiment highlighted several limitations of the model, particularly in handling concurrency issues and ensuring the correctness of formal specifications.

D. Summary of Experiments

Table I summarizes the performance of different AI models on the co-synthesis of C code, formal models in SMV, and the associated functors that establish a mapping between the two. The table compares the number of iterations required to generate correct code and models, the quality of the functors, and the success in model checking for each AI model. GPT-4o from OpenAI required 2 iterations for C code and 4 iterations for SMV model synthesis, producing high-quality functors and passing the model checking of temporal logic properties. Claude-3.5 Sonnet from Anthropic achieved the results in just 1 iteration for both C code and SMV models with similarly high-quality outputs and successful model checking.

V. Conclusions, Limitations, and Future Work

In this paper, we presented a new approach for the co-synthesis of code, their formal models, and functor-based mappings between them using Large Language Models (LLMs). Our methodology leverages the power of LLMs to generate executable code, formal models, and functors that establish a clear, human-auditable relationship between the code and its corresponding formal model. By using category theory and functors, we were able to create a systematic mapping to a formal model that enhances human trust in the synthesized code by ensuring that the formal model adheres to critical correctness properties through model checking. Using GPT-4o and Claude 3.5 Sonnet, we demonstrated the effectiveness of our approach by successfully synthesizing both C code and formal models that passed temporal logic specifications.

Our approach carries the risk of enhancing human trust in flawed code, as the functor may inadvertently cause a human to align the code with an incorrect model. This misalignment can result in misplaced confidence in the code, as it may not properly align with the verified formal model, potentially leading to a false sense of assurance.

The analysis of the functor mappings is performed manually in this paper exposing the step to potential human errors. For future work, we aim to develop automated tools that can validate these mappings, reducing the need for human intervention and increasing the efficiency of the co-synthesis process.

VI. Acknowledgments

The authors acknowledge support from DARPA award HR00112490420 and NSF 2404036. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the Department of Defense or the United States Government.

References

[1] S. Jha, J. Rushby, and N. Shankar, "Model-centered assurance for autonomous systems," in *Computer Safety, Reliability, and Security: 39th International Conference, SAFECOMP 2020*, 2020, pp. 228–243.

[2] S. Mac Lane, *Categories for the Working Mathematician*. Springer, 1971.

[3] D. I. Spivak, *Category theory for the sciences*. MIT press, 2014.

[4] S. Awodey, *Category theory*. Oxford University Press, 2010.

[5] S. K. Jha, S. Jha, R. Ewetz, and A. Velasquez, "Solving mystery planning problems using category theory, functors, and large language models," in *Proceedings of the 3rd International Conference on Assured Autonomy (ICAA)*, Nashville, TN, 2024.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, *et al.*, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[8] C. Spiess, D. Gros, K. S. Pai, M. Pradel, M. R. I. Rabin, S. Jha, P. Devanbu, and T. Ahmed, "Quality and trust in llm-generated code," *arXiv preprint arXiv:2402.02047*, 2024.

[9] S. K. Jha, S. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, R. Ewetz, and S. Neema, "Counterexample guided inductive synthesis using large language models and satisfiability solving," in *MILCOM 2023-2023 IEEE Military Communications Conference (MILCOM)*. IEEE, 2023, pp. 944–949.

[10] S. Jha, S. K. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, and S. Neema, "Dehallucinating large language models using formal methods guided iterative prompting," in *2023 IEEE International Conference on Assured Autonomy (ICAA)*. IEEE, 2023, pp. 149–152.

[11] S. Jha, T. Sahai, V. Raman, A. Pinto, and M. Francis, "Explaining ai decisions using efficient methods for learning sparse boolean formulae," *Journal of automated reasoning*, vol. 63, pp. 1055–1075, 2019.

[12] L. Sarker, M. Downing, A. Desai, and T. Bultan, "Syntactic robustness for llm-based code generation," *arXiv preprint arXiv:2404.01535*, 2024.

[13] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[14] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 215–224.

[15] S. K. Jha, *Towards automated system synthesis using sciduction*. University of California, Berkeley, 2011.

[16] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 404–415.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[18] E. Clarke, A. Fehnker, S. K. Jha, and H. Veith, "Temporal logic model checking," *Handbook of Networked and Embedded Control Systems*, pp. 539–558, 2005.

[19] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[20] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 1977, pp. 46–57.

[21] S. K. Jha, E. M. Clarke, C. J. Langmead, A. Legay, A. Platzer, and P. Zuliani, "A bayesian approach to model checking biological systems," in *Computational Methods in Systems Biology: 7th International Conference, CMSB*. Springer, 2009, pp. 218–234.

[22] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[23] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.