Harnessing ChatGPT for Model Transformation in Software Architecture: From UML State Diagrams to Rebeca Models for Formal Verification

1st Zahra Moezkarimi, 2nd Kevin Eriksson, 3rd Albin Alm Johansson, 4th Alessio Bucaioni, 5th Marjan Sirjani
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
firstname.lastname@mdu.se

***Abstract*—Software architecture relies heavily on modeling techniques to describe, analyze, and verify system designs. The Unified Modeling Language is widely recognized as the de-facto standard for modeling various types of systems. However, UML's lack of formal semantics poses challenges for performing formal verification, a critical step in ensuring the correctness of architectural models. Rebeca, an actor-based modeling language, is designed to enable formal verification of concurrent reactive systems. Previous efforts to bridge UML and Rebeca through model transformations have required combining multiple UML diagrams and a deep understanding of Rebeca, limiting practical applicability.**

In this paper, we explore the potential of leveraging large language models, specifically GPT-4, to automate the transformation of UML state diagrams into Rebeca models. Using a few-shot learning approach, we investigated the feasibility of this translation process. Initial results revealed that UML state diagrams alone were insufficient for generating accurate Rebeca models. To address this limitation, we augmented the diagrams with metadata, enabling GPT-4 to produce models that required only minor corrections to be executable in Rebeca's model-checking tool, Afra. Our findings demonstrate that LLMs hold promise in facilitating model transformations for software architecture, particularly for translating UML state diagrams into Rebeca models for formal verification. While not yet fully automated, this approach significantly reduces the effort required for transformation, paving the way for further research into the integration of LLMs into model-driven engineering practices.

***Index Terms*—Model transformation, Large language model (LLM), Formal methods, Rebeca modeling language, Unified Modeling Language (UML) state diagram.**

I. Introduction

The interplay between Software Architecture (SA) and Model-Driven Engineering (MDE) has gained increasing attention due to its potential to address the challenges of designing and verifying complex software systems. Formal verification of architectural models plays a crucial role in ensuring that systems behave as intended by verifying correctness and other critical properties during the design phase. However, despite its importance, the complexity of modeling and verification processes often limits their adoption in industrial settings.

The Unified Modeling Language (UML), a general-purpose modeling language widely regarded as the de-facto standard for system modeling [1], provides versatile diagram types suited for various applications. Yet, UML lacks formal semantics, and its diagrams are often under-specified for tasks requiring formal verification [2]. This gap creates an opportunity to explore how model-driven techniques can enhance the usability and precision of UML for formal architectural verification.

One promising avenue lies in the integration of general-purpose modeling languages like UML with domain-specific languages for formal verification. An example is Rebeca (Reactive Objects Language)[1], an actor-based modeling language designed for the formal verification of concurrent reactive systems [3]. Previous research has explored translating UML diagrams into Rebeca models using extensions like ReUML [2]. However, these approaches typically rely on multiple UML diagrams as input, imposing significant complexity and requiring substantial domain-specific expertise.

The emergence of technologies such as Large Language Models (LLMs) offers a potential paradigm shift in addressing these challenges. Generative artificial intelligence, exemplified by tools like GPT-4 [4], presents opportunities to simplify model transformation processes by reducing the need for extensive input diagrams or specialized knowledge. Inspired by low-code and no-code approaches, this work investigates how LLMs can enable more user-friendly and efficient transformations of architectural models, aligning with the broader MDE goals of automation and usability.

In this paper, we present our experience using ChatGPT (the interface fror GPT models) to transform UML state diagrams into Rebeca models. Our long-term vision is to create a semi-automatic transformation process that supports architects, including those in industrial contexts, by ensuring ease of use while maintaining accuracy through model-checker-in-the-loop verification. To achieve this, we developed an iterative three-phase method:

- Dataset Preparation: We curated a dataset of UML state diagrams paired with corresponding Rebeca models, constructed manually from existing Rebeca models. This dataset serves as a ground truth for evaluation.

[1]https://rebeca-lang.org

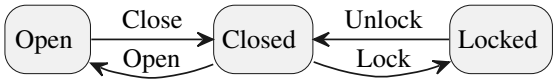

Fig. 1. UML State Diagram of a Door

- Transformation with LLMs: Using ChatGPT-4, we explored both zero-shot and few-shot learning strategies to translate UML state diagrams into Rebeca models. Metadata enhancements to UML diagrams were introduced to improve the inference of critical architectural details such as initialization and state transitions.
- Evaluation: We evaluated the generated Rebeca models using Rebeca's model-checking tool, Afra, and compared them with the ground truth.

Our results indicate that LLMs hold promise in facilitating the transformation of architectural models but are not yet mature enough for fully automated translations. Supplementary information and metadata remain essential for achieving accurate results. The main contributions of this work are as follows:

- A semi-automatic procedure for transforming UML state diagrams into Rebeca models.
- A curated dataset of UML state diagrams paired with Rebeca models to support replication and further research.
- An evaluation of the effectiveness of GPT-4[2] in performing model transformations, highlighting its strengths and limitations.
- Insights into the potential of LLMs to support MDE techniques in industrial settings, with recommendations for future improvements.

We provide a public replication package containing all the artifacts used in this work to enable independent replication and validation of our results[3]

II. Background

In this section, we provide the fundamental knowledge needed for the rest of the paper.

A. UML

UML is a general-purpose modeling language for a wide range of software systems. UML models provide visual abstractions of the system, which help identify flaws in the early stages of software development. UML diagrams can be categorized into structural and behavioral diagrams. Structural diagrams, such as class and object diagrams, represent the static structure of a system by showing its components and their relationships. Behavioral diagrams, including sequence diagrams and state diagrams, depict the dynamic behavior and interactions between system components [1]. This paper focuses on UML state diagrams. State diagrams model the conditions a system or object is in at different points in time.

[2]We use the terms ChatGPT-4 and GPT-4 interchangeably throughout the paper.

[3]Replication package: https://github.com/gnowin/UML-To-Rebecca-Dataset

UML state diagrams include various components, but for simplicity, our translations only involve the following components: states, initial states, transitions, transition triggers, and transition effects. We kept the number of UML components to a minimum to avoid unnecessary complexity. Note that the translations in this work use state diagrams augmented with some metadata, which we will describe in Section V. Figure 1 shows an example of a standard UML state diagram, representing the states of a door that cannot be opened when in the "Locked" state and cannot be locked when in the "Open" state. To use state diagrams as input for ChatGPT-4, the input needs to be text-based. To this end, we used PlantUML, an open-source modeling tool [4] that supports both textual and graphical syntax. Listing1. shows the state diagram of Figure 1 using PlantUML's textual syntax.

B. Rebeca

Rebeca is an actor-based language designed for modeling and formal verification of concurrent reactive systems [3], [5]. Rebeca features a Java-like syntax and includes reactive objects known as *rebecs*. Rebecs are concurrently executing objects and are the fundamental components of Rebeca models [5]. In Rebeca, computations occur through asynchronous *messages* between rebecs. Rebecs can only receive messages from other rebecs if they are defined as *knownrebecs* in the class definition of the sending rebec (*reactiveclass*). When a rebec sends a message to another rebec, it is received in its message queue. In a pure actor model, the queue size would be unbounded, but for model checking purposes, a limit must be set. Messages in Rebeca are analogous to method calls; when a message is dequeued, the corresponding message server (*msgsrv*) executes its code

The foundation of Rebeca is called Core Rebeca with constructs for modeling concurrent systems. However, several extensions exist for specific purposes. Timed Rebeca is introduced for real-time systems and includes the following timing primitives: *delay*, to model the passage of time during execution; *after*, to delay the arrival time of a sent message; and *deadline*, indicating that the related message must be served within a specified time frame [6].Among other extensions, which are outside the scope of this paper, we can mention

[4]https://plantuml.com

[5]Since Rebecca is a modeling language with a Java-like syntax, throughout this paper, we may use the two words Rebecca model and Rebecca code interchangeably.

```
1  @startuml
2  hide empty description
3  state Open
4  state Closed
5  state Locked
6  Open -> Closed : Close
7  Closed -> Locked : Lock
8  Locked -> Closed : Unlock
9  Closed -> Open : Open
10 @enduml
```

Listing 1: An example of a UML state diagram specified in PlantUML

Hybrid Rebeca [7] for cyber-physical systems and PTRebeca [8] for probabilistic timed systems analysis.

Afra is an Integrated Development Environment (IDE) designed to incorporate Java artifacts from projects associated with Rebeca[6]. The tool offers model development, model-checking capabilities, property specification, and visualization of counterexamples.

C. ChatGPT

Large Language Models (LLMs) are generative mathematical models that can generate and comprehend human-based text [9]. One prominent example is Chat Generative Pre-Trained Transformer, or ChatGPT [7], which functions as a virtual assistant primarily used for text generation and natural language understanding. On March 14, 2023, OpenAI introduced a newer model, GPT-4 [8]. This upgraded version of GPT provides users with advanced features and capabilities, including the ability to generate images and engage in chats using images and voice.

Designing and optimizing inputs to generative models is known as prompt engineering [10]. The goal of prompt engineering is to structure inputs so that AI models can better interpret and understand the desired tasks [9]. Prompts can be provided in various formats, but in our case, we use text prompts that include both instructions for the LLM and code examples of translations.

Zero-shot learning attempts to solve tasks without any prior training examples of that specific task [11]. For an LLM like GPT, zero-shot learning involves giving a prompt that describes what you want it to generate without providing examples of how to do it. Few-shot learning, on the other hand, is a machine learning technique that aims to train models with a small amount of training data [11]. While GPT performs well on general tasks, it can struggle with specific tasks [12]. Acquiring a large and suitable dataset for training can be resource-intensive. Few-shot learning minimizes the effort and cost needed by training LLMs on smaller datasets, allowing for more flexible and adaptable machine learning systems. Given the limited number of examples, the dataset for our experiment is small. Therefore, utilizing the few-shot learning technique for GPT-4 is essential.

III. Related Work

In this section, we present and discuss related works to highlight their results, limitations, and how they contributed to our research. Due to the widespread use of UML, many attempts at formalizing it have been made. Most of these efforts employed systematic solutions, focusing primarily on other UML diagrams, such as class diagrams and sequence diagrams, rather than on state diagrams (e.g., [13], [14]). For formalizing UML state diagrams, a recent survey [15] provides insight into the approaches that attempt to either translate state diagrams into a modeling language with formal semantics or into an intermediate language that can be converted into the input language of model checkers such as Spin and Uppal. This survey offers a comprehensive comparison based on the feature set of UML state diagrams. None of the approaches surveyed support all feature sets, a few support timing aspects, some have tool support, but most are for outdated versions of UML, highlighting the lack of a comprehensive, up-to-date approach.

There are works that attempt to translate UML to Rebeca models. Sirjani and Alavizadeh proposed a method for verifying reactive systems using the Rebeca modeling language [16]. Their approach introduced a UML profile to represent distributed systems with reactive objects and asynchronous communication, and a systematic method for generating Rebeca code from UML models. In 2007, the UML profile was extended for large systems and subsystem communication, resulting in the ReUML tool [2]. Our research also investigates translating UML to Rebeca, but focuses specifically on UML state diagrams using ChatGPT-4.

Djukanovic researched and proposed a conceptual mapping of UML to Rebeca, providing corresponding UML concepts for most elements in the Rebeca language [17]. Although the mapping aimed to facilitate the creation of an automated translation tool, Djukanovic did not implement one. Our work focuses on translating UML state diagrams to Rebeca models, while Djukanovic's work primarily addresses UML sequence, class, and object diagrams. Despite the different focuses, Djukanovic's mapping has been a valuable resource, offering useful insights into translations between UML and Rebeca.

Another relevant piece of work was published by Bucaioni et al., investigating the potential of ChatGPT to replace human programmers using a dataset of 240 programming problems in Java and C++ from LeetCode [18]. The results showed that ChatGPT performed well on easy and medium problems but struggled with harder ones, leading to the conclusion that it cannot fully replace human programmers at present. This study provides valuable insights into ChatGPT's capabilities as a programming tool, though its performance on more complex Rebeca examples remains uncertain.

IV. Research Process

In this section, we describe the research process implemented in this work that consists of three phases. During phase one, dataset creation, we collected a dataset comprising UML state diagrams paired with corresponding Rebeca models. Phase two, experimentation, involved conducting the experiment, generating Rebeca models from UML state diagrams using ChatGPT-4 with a few-shot strategy. In phase three, analysis, we analyzed the Rebeca models generated by ChatGPT-4. All the process artifacts are available in a public replication package to enable independent replication and validation of our results [3].

[6]https://rebeca-lang.org/alltools/Afra

[7]https://openai.com/

[8]https://openai.com/index/gpt-4-research/

[9]https://github.blog/2023-07-17-prompt-engineering-guide-generative-ai-llms/

Rebeca Example	Type	Reference
Dining Philosophers	Core	Rebeca Homepage[10]
Producer Consumer	Core	Rebeca Homepage
LCR Leader Election	Core	Rebeca Homepage
Sender Receiver	Core	Rebeca Homepage
Ticket Service	Timed	Handbook [19]
Train Door Controller	Timed	Paper [20]
Train-Bridge Controller	Core	Rebeca Homepage

TABLE I
AN OVERVIEW OF THE REBECA EXAMPLES CHOSEN FOR THE DATASET

A. Dataset creation

To prepare our dataset, we needed pairs of UML state diagrams and Rebeca models as ground truth. Since such a dataset did not exist, we created it by starting with existing Rebeca models and manually generating corresponding UML diagrams. We selected Rebeca examples from research papers and the Rebeca website and verify them in Afra. Some models had minor syntax errors, deadlock or queue overflow, as they were created with an older version of Rebeca or intended to indicate an error. We fixed the errors before proceeding with the translation process. Then, we manually translated Rebeca models into UML state diagrams to obtain the ground truth. Since there was no established translation procedure from Rebeca modes to UML state diagrams, we developed our own approach. To ensure feasibility, we also created a conceptual mapping of UML state diagrams to Rebeca, detailed in Section V. During these processes, we realized that UML state diagrams alone were insufficient to derive correct Rebeca models. To address this, we extended UML state diagrams with metadata, which enabled the generation of Rebeca models. We discuss medatada in Section V. Our dataset contains seven Rebeca examples (core and timed Rebeca) with corresponding translations of UML state diagrams.

Table I provides an overview of the dataset, listing the name of the example of the Rebeca models, the Rebeca type (extension), and the reference to the Rebeca example. We have detailed one of the examples in Section V.

B. Experimentation

In this phase, we conducted the experiment to translate UML state diagrams into Rebeca models using few-shot learning with ChatGPT-4. When interacting with ChatGPT-4, to keep the history clean, we disabled the ”improve the model for everyone” option to ensure that ChatGPT-4 could not learn from previous attempts. UML state diagrams from PlantUML were prompted into the chat session with the intention of ChatGPT-4 generating a corresponding Rebeca model in response. The output from ChatGPT-4 was then analyzed. Our approach for the analysis phase is detailed in Section IV-C. Each prompt and generated output response from ChatGPT-4 is documented in our GitHub repository. Listing 2 provides a sample of how the prompts are structured.

In the few-shot learning prompt, we provide multiple Rebeca examples along with their corresponding UML state diagrams. This is done until the last input and output section, where only the input PlantUML is given, and the output remains empty.

[10]https://rebeca-lang.org/Rebeca

- Lines 1-5 of Listing 2: we start with instructions that include both UML state diagrams and their corresponding Rebeca models. This helps ChatGPT-4 learn the structure and pattern of the translation.
- Multiple pairs of UML state diagrams and Rebeca models are provided in a few-shot learning approach. Each pair helps reinforce the pattern and logic of translation for ChatGPT-4.
- Lines 7-11 of Listing 2: The final instructions consist only of a UML state diagram in PlantUML format. ChatGPT-4 is expected to generate the corresponding Rebeca model based on the patterns learned from the previous examples.

C. Analysis

After conducting the experiment, we performed a comparative analysis on the generated Rebeca models. For the analysis we consider the following steps: i) Compilation check and error handling: first, we check the compilability of the generated code in Afra; in cases where there are compilation errors, we fix them; ii) Model verification: if the model compiles successfully, we then perform model checking to verify it; and iii) Comparison with ground truth: we compared the differences between the generated Rebeca code and the original code in the dataset (ground truth), in both cases of successful and unsuccessful compilation. For the comparison, in addition to an overall evaluation based on Rebeca concepts, we also perform a line-by-line comparison and a weighted analysis to provide a quantitative measure for evaluating the translation process. The results and their analysis are provided in Section VI. Note that the models may be syntactically different yet semantically equivalent.

V. MAPPING UML STATE DIAGRAMS TO REBECA MODELS

One key task in our process involved manually translating Rebeca models into UML state diagrams to create a comprehensive dataset. Our translations were inspired by the conceptual mapping of UML to Rebeca provided by Djukanovic [17] and the work of Sirjani et al. [2] on their tool for translating a modified version of UML to Rebeca code. However, they used different diagrams in their translations, so we developed our own translation process specifically for state diagrams. During the translation process, we realized that UML state diagrams alone were insufficient to produce accurate Rebeca models, and that we needed to add additional data to the

```
1  Could you translate this PlantUML diagram to Rebeca
   ↪  (Reactive Objects Language) code?
2  Input:
3  {PlantUML diagram}
4  Output:
5  {Corresponding Rebeca code}
6  ...
7  Could you translate this PlantUML diagram to Rebeca
   ↪  (Reactive Objects Language) code?
8  Input:
9  {PlantUML diagram}
10 Output:
11 {EMPTY}
```

Listing 2: Example of a Prompt for Few-Shot Learning

UML state diagrams to obtain certain behaviors, such as message servers, messages, and timing primitives. We refer to this added information as metadata. Below is the translation procedure defined by the set of rules (R) we developed:

- **R1.** Each reactive class in the Rebeca code corresponds to one UML state diagram. Regardless of the number of reactive objects (*rebecs*) initiated in the main section of Rebeca, there is only one UML state diagram for that reactive class.
- **R2.** Each state in the UML state diagram corresponds to a possible combination of values of the state variables (*statevars*) that affect the state, i.e., state variables used in conditionals.
  - The state entered from the initial state is defined by the values the state variables (*statevars*) are set to in the constructor.
- **R3.** Message servers (*msgsrv*) of a reactive class are translated into transition triggers in the corresponding UML state diagrams. A message server can act as a transition trigger for more than one transition.
- **R4.** The transition effects use metadata in the translations. The transition effect is represented as metadata encapsulated in braces. Different kinds of metadata in the transition are listed and separated, all written in a syntax similar to Rebeca code. The metadata consists of the following:
  - Conditional statements in the Rebeca code that do not relate to state variables (*statevars*) present in the UML states are stated at the beginning of the metadata. For example, a conditional about which rebec is the sender of the message.
  - Message calls to known rebecs' message servers are part of transition effects.
  - The transition effect from the initial state corresponds to the messages sent in the constructor.
  - Timing primitives are also translated into the transition effect. The *delay* and *after* primitives are added in the same order as in the Rebeca code. The *deadline* primitive was not considered in our current translation.

By applying the above procedure iteratively, we defined a conceptual mapping (Table II) of UML state diagrams to Rebeca models. This mapping is based on state diagram components, Rebeca concepts, and insights gained from iteratively refining our translation process to prepare the dataset. Each state diagram corresponds to one reactive class. For example, if we have three state diagrams, we define three corresponding reactive classes in the Rebeca model. Each state contains a subset of state variables. For instance, an actor's id may not be present in the state diagram, while it is represented as a state variable in the corresponding Rebeca model. The initial pseudo state is used to initiate the behavior of the system in the constructor. Since rebecs use messages for communication, their states change with messages. Thus, transitions are labeled with the names of message servers, and what could happen upon receiving a message is the transition effect. To keep the state diagrams simple, we do not use choice pseudo-state, state action; and final state is equivalent to a deadlock in Rebeca.

```
reactiveclass Node(8){
  knownrebecs {
    Node rightNode;
  }
  statevars {
    boolean isLeader;
    int myNumber;
    int currentLeader;
  }
  Node(byte n) {
    myNumber = n;
    currentLeader = n;
    isLeader = false;
    self.send();
  }
  msgsrv ImLeader() {
    self.ImLeader();
  }
  msgsrv send() {
    rightNode.receive(currentLeader);
  }
  msgsrv receive(int n) {
    if (n == myNumber) {
        isLeader = true;
        self.ImLeader();
    }
    else {
        if (n > currentLeader) {
            currentLeader = n;
            self.send();
        }
    }
  }
}
main {
  Node node0(node2):(4);
  Node node1(node0):(20);
  Node node2(node1):(10);
}
```

Listing 3: Rebeca Code of LCR Leader Election Example

To exemplify the mapping, we provide the file contents of the LCR Leader Election example. LCR Leader Election problem can be described as follows: within a ring network of nodes, each node initially assumes itself as the leader and communicates with its neighbor nodes to determine the actual leader. The node with the greatest ID should be the leader. Nodes exchange messages with their IDs, and if a node receives an ID greater than its current leader ID, it updates the leader's ID and notifies its neighbors. If a node receives an ID equal to its own, it is set as the leader. This process continues iteratively until only one node, with the greatest ID, remains the leader. The code for this example is shown in Listing 3. In the main section, three nodes are instantiated with IDs 4, 20, and 10. Each node has two states: leader or not leader by setting the variable *isLeader*. The code consists of one reactive class (*reactiveclass*) called "Node," which has a queue size of 8.

One property can state that eventually, node1 will be the leader, while neither *node0* nor *node2* will be the leader. Since node1's ID is greater than those of the other two nodes, the property is satisfied. When creating the UML state diagram, we considered the state variable (*statevars*) *isLeader* as the label for the states within the node, as it changes the states

UML State Diagram Concepts	Corresponding Rebeca Concept
Events	Messages received in the message servers (*msgsrv*).
States	A combination of some of the state variables (*statevars*).
Initial Pseudostate	In itself it could represent a reactive object (*rebec*) before initiated. However, the transition from Pseudostate to the state it is connected to can contain the messages sent in the constructor.
Final State	A Rebeca model that goes through formal verification should not terminate to a final state. A final state can be considered as a deadlock in the Rebeca model.
Compound/Composite States	Substates not applicable for simple translations.
Choice Pseudo-State and Guards	Can be used for conditions in Rebeca.
Transition	When message reaches message servers (*msgsrv*) and state variable (*statevars*) changes.
Self Transition	Message server (*msgsrv*) reached and no statevars (*statevars*) changed.
Transition Trigger	Message servers (*msgsrv*).
Transition Effect	The execution of what is defined in the message server (*msgsrv*) body.
State Action	Could abstractly define what happens when states are reached and exited.

TABLE II
CONCEPTUAL MAPPING BETWEEN UML STATE DIAGRAM AND REBECA

of the node. The *send()*, *receive()*, and *Imleader()* message servers (*msgsrv*) are considered labels for the transitions. Listing 4 contains the code for the PlantUML file, and Figure 2 is a visualization of the UML state diagram. All samples in the dataset include these parts. The transition from the initial state to *N_A* uses *self.send()* as metadata, initiating communications in the constructor of *Node*. The data in the transition triggers and effects is considered metadata. In the *N_A* state, there are two self-loops based on the value of the *currentLeader* variable and an outgoing transition to *N_B* if the number equals *myNumber*. The *N_B* state indicates that the node is the leader.

```
@startuml
hide empty description
state Node{
 state N_A : !isLeader
 state N_B : isLeader

[*] -> N_A : {self.send()}
 N_A -> N_A: recieve(int n) \n {n > currentLeader \n
 ↪  currentLeader = n \n self.send()}
 N_A -> N_A : send() \n
 ↪  {rightNode.recieve(currentLeader)}
 N_A --> N_B : recieve(int n) \n {n == myNumber \n
 ↪  self.ImLeader()}
 N_B -> N_B : self.ImLeader() \n {self.ImLeader()}
}
@enduml
```

Listing 4: PlantUML File of LCR Leader Election

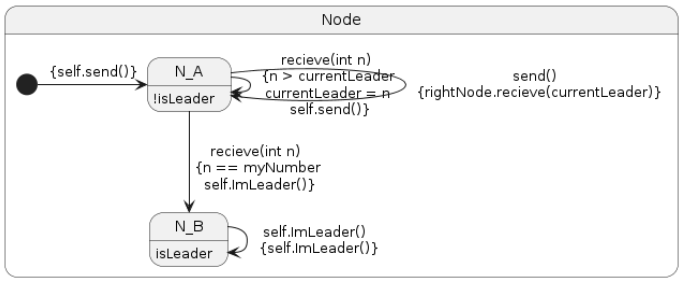

Fig. 2. UML State Diagram of LCR Leader Election

VI. GENERATING REBECA CODE FROM UML STATE DIAGRAMS WITH CHATGPT

We conducted the experiment with two different settings. In the first setting, we prompted ChatGPT-4 with two examples of UML state diagrams transformed into Rebeca models, as shown in Listing 2. In the second setting, we increased the number of examples in the prompt to five to explore how a larger training set would affect the results. Additionally, in the second setting, we asked ChatGPT-4 to translate two UML state diagrams in a single prompt. Once the Rebeca model was generated, we first compiled it in Afra to check for any compilation errors. If the model compiled successfully, we proceeded with model checking. Following successful compilation and model checking, we compared the generated Rebeca model with the original model from the dataset to identify any differences. If the generated model did not compile initially, we compared it with the original Rebeca model, fixed the errors, and then performed model checking to verify the corrected generated models.

First, we used the Dining Philosophers and Train Bridge Controller as our training examples in the prompt and asked ChatGPT-4 to generate the Rebeca model for the UML state diagram of the LCR Leader Election. The result of this iteration was that the generated code was not functional as it contained some errors. In the generated code for LCR Leader Election example, three lines of code needed fixing. Additionally, the instantiation of the rebecs in the main section did not conform to Rebecca's syntax. Some of the if statements are done slightly differently but essentially achieve the same result. A detailed explanation are provided in our replication package[3]. Then, in another setting we expanded the training set to include the following five examples: Train Door Controller, Dining Philosophers, LCR Leader Election, Train Bridge Controller, and Consumer Producer. Additionally, we asked ChatGPT-4 to generate the Rebeca models for the Sender Receiver and Ticket Service examples in the same prompt. Detailed results of these attempts can be found in our replication package[3]. None of the generated Rebeca models were successfully compiled in Afra. While the Rebeca model for the Ticket Service had only a few minor syntactic errors, the Rebeca model for the Sender Receiver example contained concepts that did not exist in Rebeca, which ChatGPT-4 added erroneously. In both cases, aside from these errors, the queue

size and overall code were acceptable, as the code worked after fixing these issues.

The common findings from the analysis of the generated models are as follows:

- The correct number of message servers (*msgsrv*) is present in the code, with names similar to those in the original code.
- ChatGPT-4 successfully initializes the correct number of rebecs in the code with accurate names.
- ChatGPT-4 added environment variables in the generated code, even though they were not necessary. This inclusion might be explained by the presence of environment variables in only one training example, which are not represented in our UML state diagrams.
- Since queue size is not specified in the UML state diagrams, ChatGPT-4 guesses this attribute, resulting in discrepancies with the target code.
- ChatGPT-4 demonstrates flaws in handling non-deterministic values, possibly because none of the training examples included non-deterministic concepts from Rebeca.
- There are flaws in recreating the ”main” section of the code. Given that the code and concepts from the ”main” section are not represented in the UML state diagrams, it was expected that this section would contain errors.

According to the results, most errors, particularly in the main section, stem from a lack of understanding of the correct syntactical structures. For example, in the case of message queues, the initial attempts by ChatGPT considered a low number, which could potentially cause queue overflow. However, after increasing the training examples to five, the numbers became acceptable. In some aspects, the results deteriorated, as the final attempts introduced new concepts from the training set, such as environment variables, that were not present in the original models (although these additions did not necessarily cause errors). It is also worth noting that ChatGPT-4 made some improvements, such as condensing four lines into one for flipping a value, and removing a ’knownrebec’ that was never used in a reactive class, demonstrating ChatGPT’s grasp of the concept of ’knownrebecs’.

In addition to the overall evaluation based on Rebeca concepts, we also performed a line-by-line comparison to provide a quantitative measure[11]. This includes analyzing the number of lines that required changes and the parts of the code that were improved. Although a line-by-line comparison can be challenging due to potential differences in code formatting that do not affect functionality, we employed a straightforward strategy to intuitively assess the results. This approach involves distinguishing the types of differences that may exist between the original Rebeca model and the generated Rebeca code for each line of code:

- *Correct (Cr)*: LoC that are accurate and do not require any modifications.

[11]Note that we do not consider curly brackets as a line of code (LoC).

```
1  env boolean networkReliability = true;  //-> Added
   ↪  (0)
2  reactiveclass Messenger(5) {
3      knownrebecs { ... }
4      statevars { ... }
5      Messenger() {
6          hasSucceeded = false;        //-> Added (0)
7          ... }
8      msgsrv sendMsg() {
9                       //-> NotExist(-2)      if
                        ↪  (hasSucceeded == true) {
10                      //-> NotInPlace(-1.5)   if
                        ↪  (sendBit == true){ sendBit =
                        ↪  false; }
11                      //->                      else {
                        ↪  sendBit = true; }}
12         medium.pass(sendBit); }
13     msgsrv receive(boolean status) {
14         hasSucceeded = status;
15         sendBit = !sendBit;          //->
           ↪  Improved(+1)
16         self.sendMsg();}
17 }
18 reactiveclass Receiver(2) {
19     knownrebecs {
20         //-> Improved(+1)  Removing an unused known
           ↪  reference, i.e., Medium medium;
21         Messenger messenger; }
22     statevars { ... }
23     Receiver() { ... }
24     msgsrv receiveMsg(boolean msgBit) { ... }
25 }
26 reactiveclass Medium(3) {
27     knownrebecs { ... }
28     statevars { ... }
29     Medium() { ... }
30     msgsrv pass(boolean msgBit) {
31         passMessage = (Math.random() > 0.5) ? true :
           ↪  networkReliability; //-> Incorrect(-1)
           ↪  passMessage = ?(true,false)
32         ... }
33 }
34 main {
35     Medium medium():();  //-> Incorrect (-1)  Medium
       ↪  medium(receiver, messenger):();
36     Messenger messenger(medium):();
37     Receiver receiver(messenger):(); //-> Incorrect
       ↪  (-1) Receiver receiver (medium,
       ↪  messenger):();
38     medium.setKnownRebecs(receiver, messenger); //->
       ↪  Added (-1)
39 }
```

Listing 5: ChatGPT-4 Generated Rebeca model for the Sender Receiver Example

- *Incorrect (Incr)*: LoC that are syntactically incorrect but located correctly within the code structure.
- *Added (Ad)*: LoC added by ChatGPT that may impact functionality or could be neutral (e.g., setting a default value for a state variable to false in the constructor).
- *Improved (Imp)*: LoC that represent enhancements over the original version.
- *Not In Place (NInP)*: LoC that are correct but misplaced within the code structure.
- *Not Exist (NE)*: Missing LoC, such as the use of ”?” for representing non-determinism in Rebeca.

In Listing 5, we depict the generated code for the Sender Receiver example, highlighting the differences from the original Rebeca model and indicating the types of discrepancies. We selected this example as it contains a diverse range of errors, making it particularly illustrative for our analysis.

Example	LoC	Cr (+1)	Incr (-1)	Ad (0/-1)	Imp (+1)	NInP (-1.5)	NE(-2)	%Cr	%WSucess
Sender Receive	48	39	3	2/1	2	1	1	85	69
Ticket Service	31	24	3	4/0	0	0	0	77	67

TABLE III
QUANTITATIVE ANALYSIS FOR CHATGPT GENERATED MODELS

The quantitative results, based on line-by-line differences, are presented in Table III.

For the Sender Receiver example, ChatGPT-4 generated 39 correct lines out of 48, achieving an 85% success rate. The Ticket Service shows a success rate of 77%. However, a more nuanced analysis might benefit from a weighted approach. Consider the following simple weighting mechanism: assign +1 for each *Correct* line, -1 for each *Incorrect* line, 0 or -1 for *Added* lines depending on their impact, +1 for *Improved* lines, -1.5 for *Not In Place*, and -2 for *Not Exist*. Using this method, the weighted success rates for the generated Sender Receiver and Ticket Service codes are 69% and 67%, respectively. The last two columns of Table III show the percentage of correct lines of code and the weighted success rate for each example.

VII. DISCUSSION

In this work, we shared our experience in generating Rebeca models from UML state diagrams using ChatGPT-4 and few-shot learning. To prepare the dataset including Rebeca models and their corresponding UML state diagrams, we introduced a mapping from UML state diagrams to Rebeca models. Several challenges emerged during the translation process, providing insight that we can not represent all that is captured in a Rebeca model using only UML state diagrams. This led us to augment the state diagrams with metadata to enable a more accurate translation. Examples of concepts in Rebeca that lack equivalent counterparts in UML state diagrams include the main function and its contents, environment variables, state variables in message servers, timing primitives, constructors that builds the initial behavior of the model, and message calls to other rebecs. We included a subset of the missing elements in state diagrams as metadata that we believe ChatGPT cannot infer and, have higher priority for describing the behavior of the system. This subset includes message calls to other rebecs, timing primitives, initialization messages in the constructor, and certain conditional statements.

The Rebeca models generated through few-shot learning could not compile in Afra, as they either contained syntax errors or non-existing concepts in Rebeca. According to the results, most of the errors stem from a lack of understanding of how they should be written syntactically. A potential future research direction could investigate the necessary and sufficient metadata required to generate correct and efficient Rebeca models from UML diagrams using LLMs. Increasing the training examples to five led to improvements, such as acceptable message queue values. However, some outcomes were less favorable, as the final models incorporated new concepts from the training set, like additional environment variables, which did not necessarily result in errors. It is noteworthy that ChatGPT-4 introduced other enhancements, such as providing one line instead of four for flipping a value, or removing an unused knownrebec from a reactive class that underscore ChatGPT's improved grasp of concepts like knownrebecs.

The creation of our dataset poses several validity threats. Firstly, our dataset comprises only a few examples. Although few-shot learning is designed for small datasets, the limited number of examples might still be too small to achieve optimal results. Additionally, including both Core and Timed Rebeca models raises the concern that ChatGPT-4 might not fully differentiate between them. Another issue is dataset bias due to the addition of metadata for translation. Furthermore, the lack of transparency in how ChatGPT-4 learns from history and interactions introduces an external validity threat; although we disabled learning from our chat history, it can still learn from other online sources. Lastly, ChatGPT-4's non-deterministic behavior may lead to varying results from the same input, affecting result reproducibility.

VIII. CONCLUSIONS AND FUTURE WORK

Formal verification is crucial for ensuring systems operate as intended during design, but its complexity limits industrial application. Although UML is the standard for system modeling, it lacks the formal semantics needed for verification. Integrating UML with verification-specific languages like Rebeca could bridge this gap. We are seeking a model transformation approach that requires minimal knowledge and effort. In this work, we shared our experience using ChatGPT-4 to generate Rebeca models from UML state diagrams, revealing that UML alone does not suffice for accurate translations. Our enhancements with metadata improved model accuracy. According to the results, most errors, particularly in the main section, arise from a lack of understanding of the correct syntactical structures. For the Sender Receiver example, ChatGPT-4 generated 39 correct lines out of 48, achieving an 85% success rate. The Ticket Service example displayed a slightly lower success rate of 77%.

We conclude that using few-shot learning with ChatGPT-4 holds potential for translating UML state diagrams into Rebeca models. However, we believe that incorporating more examples and Retrieval-Augmented Generation (RAG) techniques alongside LLMs for tuning can enhance the translation process. Other potential future research directions could include exploring the essential metadata needed to generate accurate and efficient Rebeca models from UML diagrams using LLMs, providing the syntax of Rebeca along with a brief description of each example, and fine-tuning LLMs for our translation purposes. The results of this research such as

conceptual mappings and identifying key minimal elements and representations can be leveraged in other model transformations, especially those targeting actor-based languages and formal verification.

REFERENCES

[1] *OMG Unified Modeling Language (OMG UML)*, 2nd ed., Object Management Group, 12 2017, [Online] Available: https://www.omg.org/spec/UML/2.5.1/PDF.

[2] S. F. Alavizaedh, A. H. Nekoo, and M. sirjani, "Reuml: a uml profile for modeling and verification of reactive systems," in *International Conference on Software Engineering Advances (ICSEA 2007)*, 2007, pp. 50–50.

[3] M. Sirjani, "Rebeca: Theory, applications, and tools," in *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4709. Springer, 2006, pp. 102–126.

[4] S. Noy and W. Zhang, "Experimental evidence on the productivity effects of generative artificial intelligence," *Science*, vol. 381, no. 6654, pp. 187–192, 2023.

[5] M. Sirjani, A. Movaghar, and M. Mousavi, "Compositional verification of an object-based model for reactive systems," in *AVoCS 2001*, 2001. [Online]. Available: https://rebeca-lang.org/assets/papers/2001/CompositionalVerificationOfAnObject-BasedModelForReactiveSystems.pdf

[6] M. Sirjani and E. Khamespanah, "On time actors." in *Theory and Practice of Formal Methods*, ser. Lecture Notes in Computer Science, E. Ábrahám, M. M. Bonsangue, and E. B. Johnsen, Eds., vol. 9660. Springer, 2016, pp. 373–392.

[7] I. Jahandideh, F. Ghassemi, and M. Sirjani, "Hybrid rebeca: Modeling and analyzing of cyber-physical systems," in *Cyber Physical Systems. Model-Based Design - CyPhy 2018*, ser. Lecture Notes in Computer Science, R. D. Chamberlain, W. Taha, and M. Törngren, Eds., vol. 11615. Springer, 2018, pp. 3–27.

[8] A. Jafari, E. Khamespanah, M. Sirjani, H. Hermanns, and M. Cimini, "Ptrebeca: Modeling and analysis of distributed and asynchronous systems," *Sci. Comput. Program.*, vol. 128, pp. 22–50, 2016.

[9] M. Shanahan, "Talking about large language models," *Commun. ACM*, vol. 67, no. 2, p. 68–79, jan 2024. [Online]. Available: https://doi.org/10.1145/3624724

[10] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *International Conference on Data Intelligence and Cognitive Informatics*. Springer, 2023, pp. 387–402.

[11] A. Parnami and M. Lee, "Learning from few examples: A summary of approaches to few-shot learning," *arXiv preprint arXiv:2203.04291*, 2022.

[12] J. Kocoń, I. Cichecki, O. Kaszyca, M. Kochanek, D. Szydło, J. Baran, J. Bielaniewicz, M. Gruza, A. Janz, K. Kanclerz *et al.*, "Chatgpt: Jack of all trades, master of none," *Information Fusion*, vol. 99, p. 101861, 2023.

[13] R. M. Borges and A. C. Mota, "Integrating uml and formal methods," *Electronic Notes in Theoretical Computer Science*, vol. 184, pp. 97–112, 2007, proceedings of the Second Brazilian Symposium on Formal Methods (SBMF 2005). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066107004379

[14] S. Weixuan, Z. Hong, F. Yangzhen, and F. Chao, "A method for the translation from uml into event-b," in *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2016, pp. 349–352.

[15] E. André, S. Liu, Y. Liu, C. Choppy, J. Sun, and J. S. Dong, "Formalizing uml state machines for automated verification – a survey," *ACM Comput. Surv.*, vol. 55, no. 13s, jul 2023. [Online]. Available: https://doi.org/10.1145/3579821

[16] F. Alavizadeh and M. Sirjani, "Using UML to develop verifiable reactive systems," in *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*, H. R. Arabnia and H. Reza, Eds. CSREA Press, 2006, pp. 554–561.

[17] V. Djukanovic, "Mapping uml diagrams to the reactive object language (rebeca)," 2019. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-44137

[18] A. Bucaioni, H. Ekedahl, V. Helander, and P. T. Nguyen, vol. 15, 01 2024.

[19] A. Jafari, E. Khamespahanh, H. Hojjat, Z. S. Kaviani, and M. Sirjani, *Rebeca User Manual*, 2nd ed., 12 2016, [Online] Available: https://rebeca-lang.org/assets/documents/manual.2.1.pdf.

[20] M. Sirjani, L. Provenzano, S. A. Asadollah, M. H. Moghadam, and M. Saadatmand, "Towards a verification-driven iterative development of software for safety-critical cyber-physical systems," *J. Internet Serv. Appl.*, vol. 12, no. 1, p. 2, 2021. [Online]. Available: https://doi.org/10.1186/s13174-021-00132-z