FEDERAL UNIVERSITY OF PERNAMBUCO
CENTER FOR INFORMATICS
UNDERGRADUATE PROGRAM IN COMPUTER ENGINEERING

Pedro Vítor Cunha

A Formal translation from Robosim to C++ applied in the Robocup 2D simulation environment

Recife

2025

Pedro Vítor Cunha

A Formal translation from Robosim to C++ applied in the Robocup 2D simulation environment

Work presented to the Bachelor of Computer Engineering Program at the Computer Science Center of the Federal University of Pernambuco, as a partial requirement for obtaining the degree of Bachelor in Computer Engineering.

Area of Concentration: Software Engineering

Advisor: Augusto Sampaio

Co-advisor: Madiel Conserva Filho

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Cunha, Pedro Vitor.
A Formal translation from Robosim to C++ applied in the Robocup 2D simulation environment / Pedro Vitor Cunha. - Recife, 2025.
39p : il., tab.

Orientador(a): Augusto Sampaio
Cooorientador(a): Madiel Conserva
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências, apêndices.

1. Engenharia de software. 2. Métodos Formais. 3. Máquinas de estado finito. I. Sampaio, Augusto . (Orientação). II. Conserva, Madiel. (Coorientação). IV. Título.

020 CDD (22.ed.)

PEDRO VITOR CUNHA

A FORMAL TRANSLATION FROM ROBOSIM TO C++ APPLIED IN THE ROBOCUP 2D SIMULATION ENVIRONMENT

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 15/04/2025

BANCA EXAMINADORA

Profa. Dr. Augusto Sampaio (Orientador)

Universidade Federal de Pernambuco

Prof. Dr. Madiel Conserva (Coorientador)

Universidade Federal de Pernambuco

Prof. Dra. Edna Natividade(Examinador Interno)

Universidade Federal de Pernambuco

Dedico este texto aos meus pais que sempre me mostraram a beleza no ato de estudar, da dedicação e sobretudo da união.

ACKNOWLEDGEMENTS

Agradeço todos que estiveram comigo durante esses anos de graduação. Foi uma maravilhosa e rica jornada.

Antes dos agradecimentos pessoais, preciso trazer uma menção aos meus amigos e companheiros de disciplina de Tópicos Avançados em Engenharia de Software, ministrada pelo Professor Augusto, que me ajudaram com uma parte do que foi utilizado nessa dissertação ao fazer parte dos experimentos análise do que virá a ser explicado na seção de LLM. Gabriel, João e Andresa meu muitíssimo obrigado.

Agradecer também ao RoboCIn, que muito me educou durante a graduação e fortaleceu para que pudesse dar os meus primeiros pequenos passos como aspirante a pesquisador e no mercado de trabalho.

Aos meus amigos que a sua maneira foram sendo suporte nos momentos difíceis e companheiros nos momentos de alegria. Sobretudo, Rafinha que se faz presente desde a escola e trouxe toda sua leveza na minha caminhada também, Biel que veio como parceiro de estudos e se tornou um grande amigo, Jonga, Allan, Nuneira, Elisson (Elibala), Goncinha, Cris, Andresa, Dri, Binho, Thiago e graças a Deus tantos outros incríveis amigos que Ele colocou no meu caminho.

Agradecer também aos professores que me acompanharam, em especial Augusto e Madiel que tornaram o ato de pesquisar algo extremamente divertido e posso considerar como amigos. Mas também aos outros que me formaram como Divanilson e Abel pelo tempo no LIVE, Breno pelo tempo na Motorola, todos vocês fazem parte da minha educação.

Agradecer aos meu pais por todo o esforço e suporte durante toda a minha vida, essa conquista é tão minha quanto deles. Minhas irmãs que são minhas melhores amigas e escutaram tudo que podiam escutar dos meus desabafos. E a minhas avós que foram meus tesouros mais preciosos.

Por último, a Deus que escreveu meu caminho com muito carinho.

"Se a educação sozinha não transforma a sociedade, sem ela tampouco a sociedade muda."
(Freire, Paulo)

ABSTRACT

In the robotics field, simulation environments are essential for designing, evaluating, and validating complex autonomous behaviors before actual deployment, and the RoboCup 2D Simulation League provides a robust framework for testing multi-agent coordination and decision-making under uncertainty, benefiting AI and robotics research. Domain-specific languages like RoboSim enable intuitive high-level robot behavior modeling using finite-state machines (FSMs), yet converting these models into executable code is challenging due to the semantic gap between declarative DSL structures and imperative languages such as C++. This research presents an Eclipse plugin that automates the translation of RoboSim models into code using a C++ library that supports state machine implementations. The translation is systematic as it is based on defining and implementing translation rules to generate accurate FSM representations for the RoboCup 2D simulation environment. The approach ensures behavior consistency, facilitating the transition from high-level design to implementation and simplifying the development of autonomous agents while minimizing errors during deployment.

Keywords: RoboCup, simulation, finite-state machines, RoboSim, C++, autonomous agents, multi-agent.

RESUMO

Na área da robótica, ambientes de simulação são fundamentais para o projeto, a avaliação e a validação de comportamentos autônomos complexos antes da implantação real. A liga de simulação 2D do RoboCup oferece uma estrutura robusta para testes de coordenação multiagente e tomada de decisão sob incerteza, beneficiando pesquisas em inteligência artificial e robótica. Linguagens específicas de domínio, como a RoboSim, possibilitam a modelagem intuitiva e de alto nível do comportamento de robôs por meio de máquinas de estados finitos (FSMs). No entanto, a conversão desses modelos em código executável é desafiadora devido à lacuna semântica entre as estruturas declarativas da DSL e linguagens imperativas como C++ utilizando uma biblioteca que permite a implementação de máquinas de estado. Esta pesquisa apresenta um plugin para o Eclipse que automatiza a tradução de modelos RoboSim para código C++. A tradução é sistematica por se basear na definição e implementação de regras de conversão para gerar representações precisas de FSMs no ambiente de simulação 2D do RoboCup. A abordagem garante a consistência do comportamento, facilitando a transição do design de alto nível para a implementação e simplificando o desenvolvimento de agentes autônomos, ao mesmo tempo em que minimiza quantidade ou ocorrência erros durante a implantação.

Palavras-chaves: RoboCup. Simulação. máquinas de estados finitos. RoboSim. C++. agentes autônomos. multiagente.

LIST OF FIGURES

LISTINGS

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Simulation plays a vital role in the development of robotic systems (ŽLAJPAH, 2008). Before deploying robots in real-world environments, where errors can be costly or even dangerous, developers rely on simulation environments to model, test, and iterate control logic, sensor integration, and overall system behavior. Simulations enable rapid prototyping, reproducibility, and safe experimentation, making them indispensable tools in modern robotics workflows.

A notable example of simulation-driven development is the RoboCup Simulation Leagues. RoboCup[1] is an international scientific initiative focused on advancing the state of the art in intelligent robotics. Established in 1997, its original mission was to field a team of autonomous robots capable of defeating the human soccer World Cup champions by 2050.

The RoboCup Simulation Leagues serve as a long-standing research platform in which autonomous agents compete in simulated soccer matches. Each agent must perceive its environment through noisy and partial observations, making real-time decisions while coordination with teammates. This simulation environment has become a testing ground for a wide range of topics of artificial intelligence (AI), including multi-agent coordination and decision making under uncertainty (PROKOPENKO; WANG, 2019).

In the development of autonomous systems and robotic applications, the use of domain-specific languages (DSL) has become increasingly prevalent (CHALLENGER et al., 2014). These languages offer high-level abstractions that enable developers to model complex behaviors, such as decision making and control logic, in a declarative and intuitive manner. One such language, RoboSim (CAVALCANTI et al., 2019), is specifically designed to model simulations of robotic systems based on finite-state machines (FSMs). RoboSim features include diagram editing in RoboTool and CSP (BROOKES; ROSCOE, 2021) formal semantics. RoboTool automatically generates this semantics, enabling verification of classical concurrency properties and domain-specific properties.

Since it is a nonexecutable modeling language, deploying RoboSim models to real-world systems requires translation into a general-purpose programming language such as C++. This transition presents several challenges. RoboSim's abstract semantics, hierarchical states, and event-driven behavior must be faithfully mapped to an imperative paradigm, by preserving the intended logic, timing guarantees, and system constraints.

[1] https://www.robocup.org/

This dissertation explores the methodology and design principles underlying the translation of RoboSim state machines into C++ implementations. We analyze the core syntax and semantics of RoboSim, identify key translation patterns, and propose a systematic rule-based approach to achieve the code in the target language. Our approach enables developers to bridge the gap between high-level simulation and low-level deployment, ensuring that the logic developed and validated in RoboSim carries over accurately into production environments. We also investigated employing Large Language Models (LLMs) as an alternative strategy to systematic rule-based translation. Generative AI can create new content by identifying patterns in data, supporting tools that improve writing, design, and coding, thus boosting problem-solving efficiency and creativity.

The structure of the dissertation is as follows. Chapter 2 reviews related work on the translation of DSLs to other domains. Chapter 3 presents an overview of the Simulation 2D environment, and RoboSim. Chapter 4 described the modeling approach, the translation rules from RoboSim to C++, and an example of transformation setup. Chapter 5 describes the implementation details of the translation rules. Also, 6 describes an alternative solution for a translation using the model file description and GPT. Finally, we conclude our work in Chapter 7.

2 RELATED WORK

Regarding the modeling aspect, this work uses the RoboSim notation to specify simulations that are technology agnostic. RoboSim is part of an elaborate and modern software engineering process for developing robotic systems called RobStar. Apart from RoboSim for simulation and RoboTool as tool support, RoboStar offers another notation for modeling abstract designs of robotic controller software.

Robochart (MIYAZAWA et al., 2019) aims to reduce the gap between abstract design model of controller software and their verification. It is DSL for the design and formal verification of robotic controllers. RoboChart is based on UML state machines and provides constructs for modeling timed and concurrent behaviors, as well as architectural components such as controllers and robotic platforms. Like RoboSim, Its formal semantics is defined using CSP (Communicating Sequential Processes) and supports automated reasoning via model checking and theorem proving. This work highlights the importance of integrating formal semantics into robotic DSLs to bridge the gap between design and verification, an aspect often lacking the work in ad hoc state machine languages used in simulation.

The work in (ZHANG et al., 2021) proposes a transformation strategy for translating RoboSim models into the UPPAAL (GIBSON-ROBINSON et al., 2014) timed automata network (NTA), allowing formal verification of real-time, stochastic, and hybrid behaviors. This work uses pattern-based mapping rules to preserve RoboSim's cyclic execution semantics and is implemented as a RoboTool plugin. The purpose here is similar, but we aim for an executable implementation in C++, whereas that work concentrates on formal verification via translation into UPPAAL.

The work in (SANTOS; FILHO; SAMPAIO, 2023), the authors present a model-driven development approach for robotic systems targeting the RoboCup Brazil Flying Robots Trial League. Using RoboChart and RoboTool, the authors design and verify the behavior of an autonomous UAV, ensuring both classical properties (e.g., deadlock freedom) and domain-specific requirements through model checking with FDR (GIBSON-ROBINSON et al., 2014). Their work emphasizes the importance of formal modeling in competitive robotics and demonstrates how design can guide and structure implementation, particularly a Python-based runtime using state machines. This highlights the applicability of RoboChart not only for correctness assurance but also as a blueprint for generating executable architectures.

The work in (YE; FOSTER; WOODCOCK, 2023) advances robotic verification by integrating formal methods with animation through interaction trees (ITrees) for RoboChart models. Their framework provides composable operational semantics for reactive and concurrent robotic programs, validated through case studies on chemical detector and patrol robots. By resolving nondeterministic choices in state machines and generating verified Haskell representations via Isabelle/HOL, they bridge the gap between mathematical verification and practical validation. This hybrid approach offers both formal rigor and intuitive testing capabilities, addressing the reliability challenges faced by autonomous systems in complex environments.

In (ISOBE et al., 2021) the authors explore formal verification of concurrent finite-state machines for cooperative robotics through a transport robot case study. Using Communicating Sequential Processes (CSP) for specification and the FDR model checker for verification, they established a complete formal pipeline from design to implementation. Their methodology identified control logic errors before physical deployment while providing reusable formal descriptions of robot behaviors. The validated implementation on Robot Technology Middleware using Raspberry Pi robots effectively demonstrates how formal methods can enhance reliability in practical cooperative robotic systems, particularly for critical mode transitions and event-based interactions.

The work in (BAZYDŁO; ADAMSKI; STEFANOWICZ, 2014) addresses the critical bridge between high-level system modeling and hardware implementation through their work, the translation of UML state machine diagrams into Verilog. Their methodology leverages a Hierarchical Concurrent Finite State Machine (HCFSM) as an intermediate representation to decompose complex UML diagrams into master-slave FSM structures, enabling the generation of modular, synthesizable Verilog code.

3 BACKGROUND

3.1 ROBOCUP SOCCER SIMULATION 2D

The Soccer Simulation 2D League is the longest running competition within the Robocup Soccer Simulation Leagues [1]. This league only involves virtual robots and each virtual robot can have unique strategies and traits, while every virtual team is a collection of software programs (CHEN et al., 2003).

Figure 1 – 2D soccer match

Source: Author (2025)

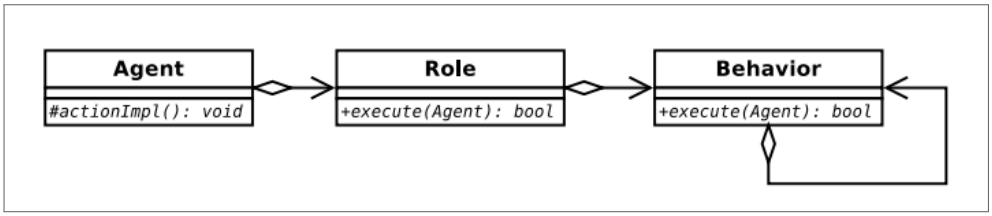
The RoboCup Soccer Simulation 2D league is centered on the RoboCup Soccer Simulator Server (RCSSS) [2] (NODA; MATSUBARA, 1996). This simulator allows two teams, each consisting of 11 independent players and an autonomous coach agent, to participate in a football match with highly realistic rules and real-time action. This platform is primarily used for research and educational purposes in the field of multi-agent systems. The RCSSS executes and manages a 2D football match and has a comprehensive knowledge of the game, including the precise location of all elements and their movements. In addition, the game relies on communication between the server and each agent. Players receive incomplete and imprecise information about the environment and, based on their logic and algorithms, the agents generate fundamental

[1] https://ssim.robocup.org/
[2] https://github.com/rcsoccersim/rcssserver

commands (such as dashing, turning, or kicking) to interact with the game world. Figure 1 shows a 2D soccer match before a penalty kick in game mode *penalty_setup_r*, indicating the penalty is occurring on the right side of the field. Both goalies are moved to the right goal area. Each match features two teams, distinguished by different colors, with 11 players per team, and each player functions as a separate process.

Several open-source code bases in C++ are available, including Gliders2D (PROKOPENKO; WANG, 2018), Cyrus2D (ZARE et al., 2022), and Helios Base(AKIYAMA; NAKASHIMA, 2014), the most well-known and used in this work. Figure 2 illustrates the architecture of all these code bases, which follow the same agent behavior structure as presented in (AKIYAMA; NAKASHIMA, 2014).

Figure 2 – Decision making of agent2d

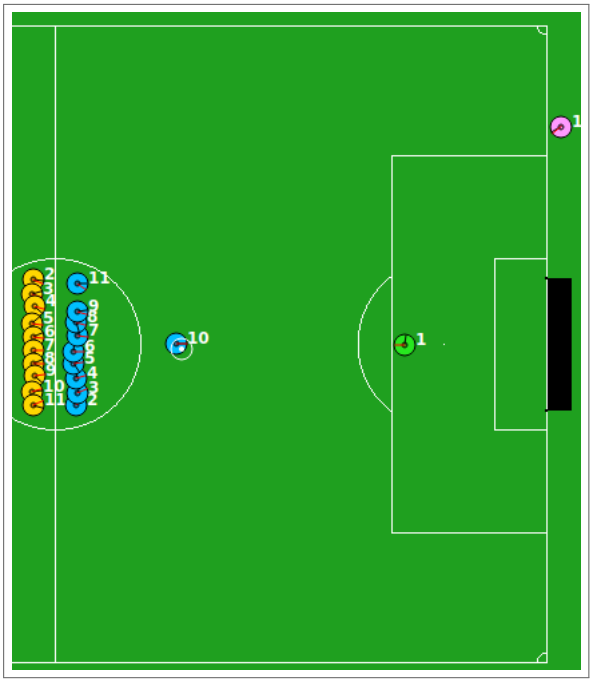


Source: (AKIYAMA; NAKASHIMA, 2014)

Our focus here is on behavior, which is a set of rules that define a complex action. A behavior can be used by multiple roles, meaning that, for example, the same behaviour used by the center forward can also be used by the wing attacker, depending on the strategy. Behaviors can be as complex as the situation requires. For example, we can create a behavior to execute the best possible pass, as well as one for a basic movement from point A to point B. A complex behavior can be composed of simpler behaviors, for example we can create a behavior to execute two actions in the same cycle, or create a behavior that executes other behaviors depending on the game situation.

Another interesting example is the behaviors of the players involved in a penalty kick, namely the goalie and the kicker, as illustrated in Figure 3. From the goalie's point of view, it starts at the goal and can move anywhere on the field. Meanwhile, the kicker begins in the center circle and tries to score a goal. As shown in Figure 3, the entire green area, bordered by the outer white line, is a walkable space for both players, while the other players do not move.

Figure 3 – Illustration of the penalty kick scenario

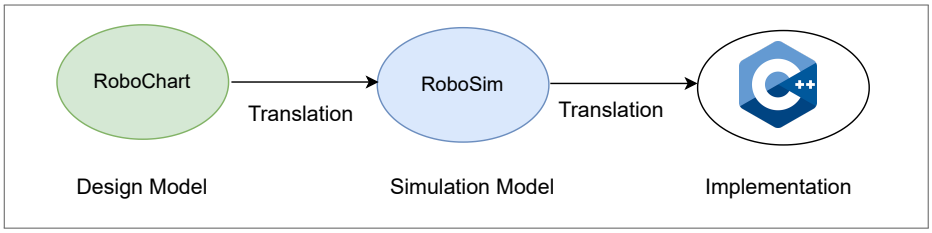


Source: Author (2025)

3.2 ROBOSIM

RoboSim is part of the RoboStar framework [3], whose central objective is to develop a systematic and rigorous methodology for modelling, verification, simulation, testing, and implementation of robotic applications. In the RoboStar technology, the first step focuses on abstract design models, as presented in Figure 4. These models are written using RoboChart, a DSL aligned with notations commonly accepted by roboticists. From a RoboChart model, we automatically generate a simulation model in RoboSim, which is the focus of this work. We start from a simulation model because our context is a simulation environment in the Soccer Simulation 2D League. This simulation model is then translated into a C++ implementation.

Figure 4 – RoboStar framework



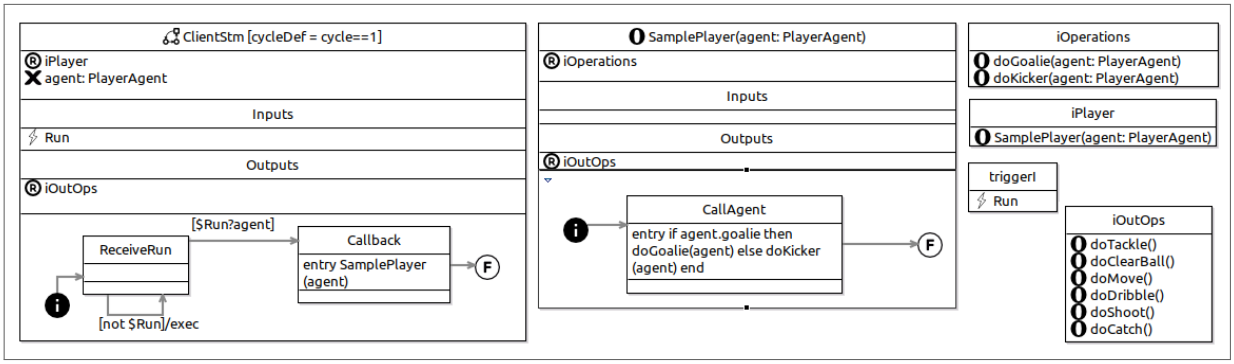
Source: Author (2025)

[3] https://robostar.cs.york.ac.uk/

RoboSim (CAVALCANTI et al., 2019) is a techonology-independent notation for modeling simulations of robotic systems. It is based on the Unified Model Language (UML) state machines to specify the behavior of the control software of a simulation. RoboSim has a formal semantics defined in tock-CSP, a version of CSP that considers time (ROSCOE, 2011). So, verification of classical behavioural properties, such as determinism, deadlock and livelock freedom of the models, can be performed by the model checker FDR (GIBSON-ROBINSON et al., 2014). Also, domain-specific properties can be proved as well.

To illustrate the notation and the main components of a RoboSim simulation, we consider the state machine ClientStm in Figure 5. A RoboSim model specifies a cyclic mechanism since it is an abstraction for a simulation. So, in the ClientStm, we note the definition of the cycle period using the predicate cycle == 1. In our example, we consider that each cycle takes one time unit.

Figure 5 – RoboSim model



Source: Author (2025)

In a RoboSim state machine definition, inputs and outputs are made explicit. For our machine in Figure 5, the event Run is an input, while the operations declared in the interfaces iOutOps are outputs. The machine ClientStm also requires the interface iPlayer, which declares the software operation SamplePlayer(), and declares a local variable agent of type PlayerAgent.

Each cycle of execution of a simulation machine is marked with the special event **exec**, which does not need to be declared. The behavior of ClientStm begins in the first cycle, where its initial transition, from the initial junction to the state ReceiveRun, is fired. From this state, there are two possible transitions. If condition not $Run holds, the self-transition of ReceiveRun is taken, and the machine starts a new cycle to read a new value from the environment and assign it to the boolean variable $Run again. Otherwise, it means that the controller behaviour must proceed, as part of the current cycle, and the control flows to the

state CallBack, where its entry action is immediately executed. In this case, it is a call to the software operation SamplePlayer. After that, the model reaches its final state.

A software operation in RoboSim is similar to a state machine, except that it can have parameters. The behavior of SamplePlayer() is as follows. The initial transition leads to the state ActionImpl, where its entry action checks whether the parameter agent is equal to goalie. If so, the software operation doGoalie() is called. Otherwise, it calls the operation doKicher(). The specifications of these two operations are omitted here. Finally, the transition from CallAgent to the final state is made.

It is important to emphasize that, while the state machine depicted in Figure 5 encapsulates the information propagated to the software operations, it is an abstraction of both a network interaction and a function invocation in our C++ implementation.

4 METHODOLOGY

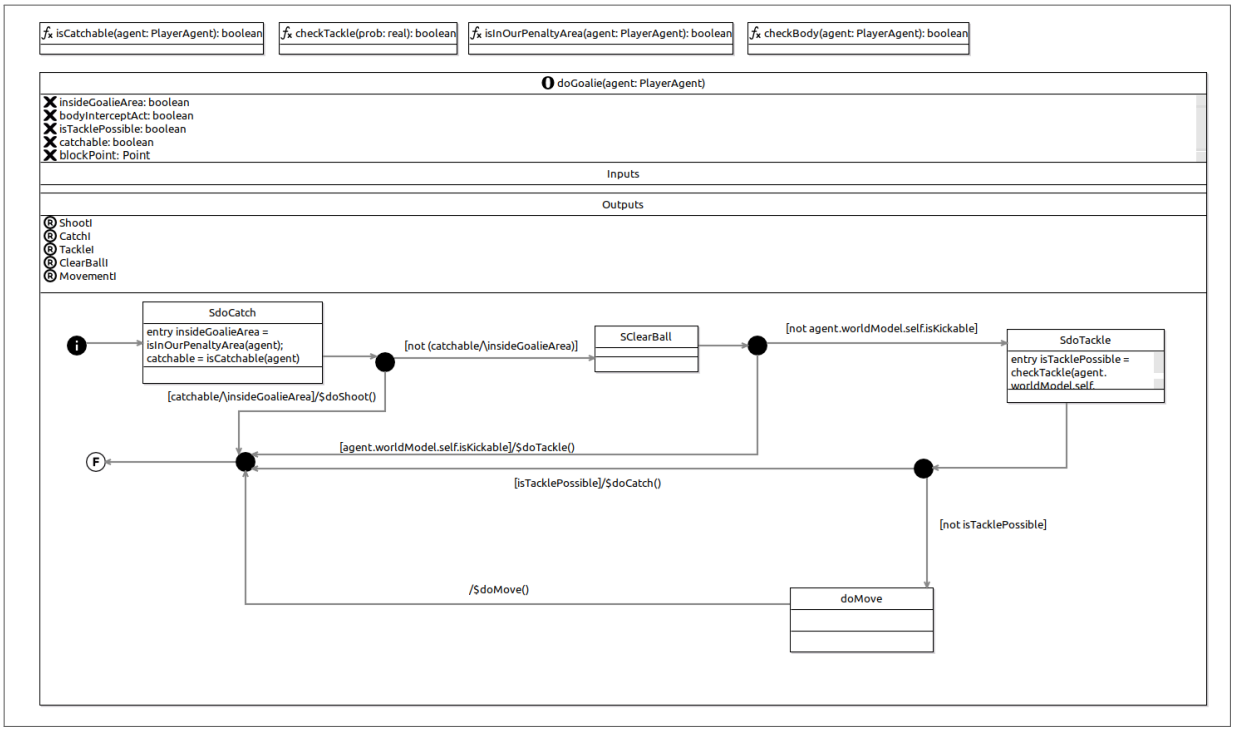
This chapter describes the methodology used in this study, detailing the techniques, tools, and procedures employed to define our translation rules. It is designed to ensure cohesion during the translation from RoboSim to C++.

4.1 MODELING

This section focuses on detailing a RoboSim model that outlines a RoboSim Software Operation.

4.1.1 Goalie Operation

Figure 6 – Goalie Behavior Model



Source: Author (2025)

In Figure 6, we present the RoboSim operation doGoalie(), which models the behavior of the goalie during a penalty kick. Its behavior starts at the initial junction and immediately transitions to the state *SDoCatch*, whose entry action uses (auxiliary) functions to compute the values required to determine whether the goalie is positioned within the designated goalie

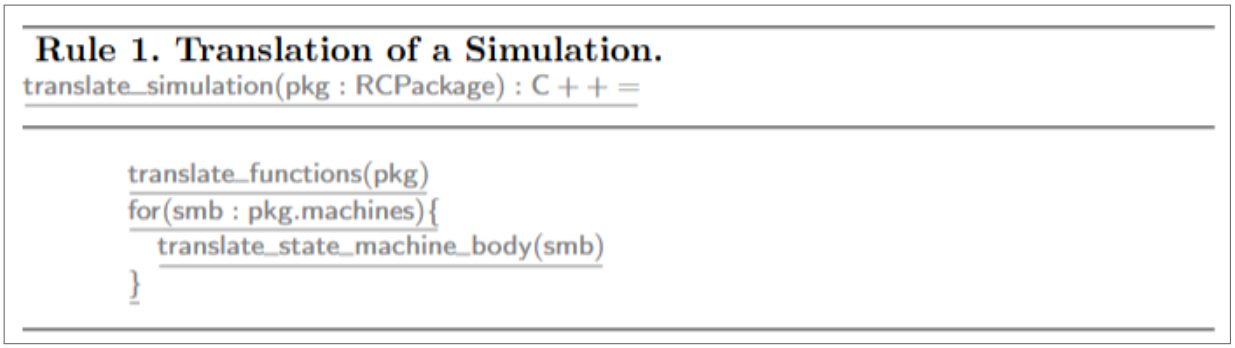
area and whether the ball is in a catchable state. If both conditions are satisfied, then the goalie is able to shoot the ball, as specified by the platform operation doShoot(), after which the control flows directly to the final state of the operation. Otherwise, the transition to the SClearBall state is fired, followed by a subsequent transition from it to a junction.

At this point, if the ball is kickable, (agent.worldModel.self.isKickable), the goalie can intercept the ball using the operation doTackle(), that is, for instance, by kicking it away. Subsequently, the model reaches its final state. If the ball is not kickable, the control flows to state SdoTackle, whose entry action determines whether a tackle is feasible. If so, the goalie catches the ball, and the execution ends. Otherwise, the goalie moves (specified by doMove()), and the operation doGoalie() reaches its final state.

4.2 TRANSLATION RULES

The mapping from RoboSim to Helios Base (a code base in C++) is defined by a collection of precise and compositional rules. The main rule, Rule 1, presented in Figure 7, converts a RoboSim simulation into a C++ implementation. To achieve this, Rule 1 takes an *RCPackage* as a parameter. This package contains all the necessary information for subsequent rules. The meta-notation used to characterize the translation process is underlined, and C++ terms are written in italics (and in different color: purple) as usual. The meta-notation is straightforward and explained as needed.

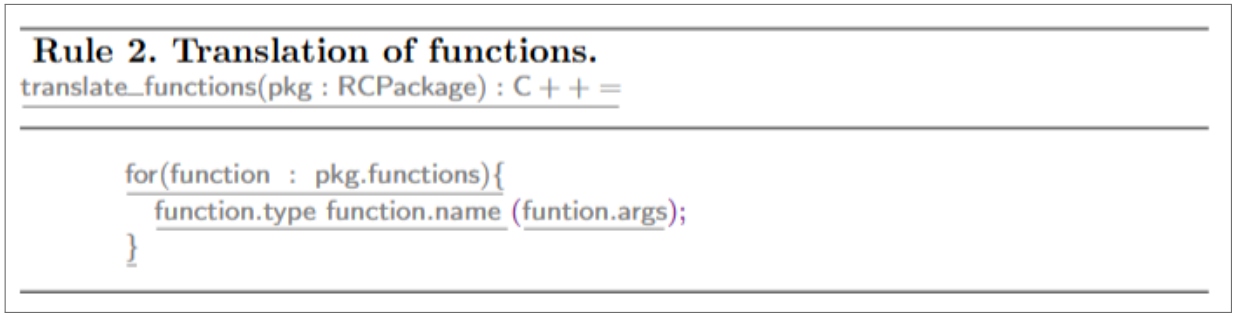
Figure 7 – Translation of a simulation



Source: Author (2025)

In Rule 1, we first translate the functions of the simulation model using Rule 2. To translate them, we iterate over all functions gathering their types, names, and arguments to convert them into the C++ syntax.

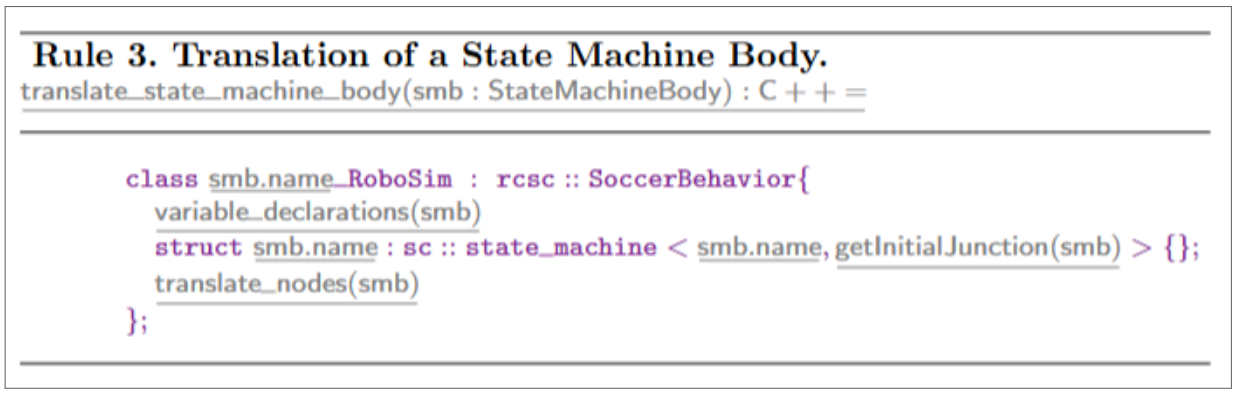
Figure 8 – Rule to translate functions



Source: Author (2025)

After translating the functions, Rule 1 deals with the translation of the simulation machines, which is effectively handled by Rule 3. This rule takes a StateMachineBody (smb) as a parameter. In RoboSim, both state machines and software operations are types of smb. In Rule 3, each smb is translated into a C++ class of type SoccerBehavior, where the smb variables become class attributes, as defined by Function 3 in Appendix A. Afterwards, Rule 3 creates a C++ structure (struct) that inherits from the state machine definition of *libboost*[1], a C++ library. This structure is instantiated using the name and initial junction of the machine. Finally, Rule 3 invokes the function translate_nodes(), defined in Rule 4, to handle the translation of the machine's nodes.

Figure 9 – Rule to translate a State Machine Body



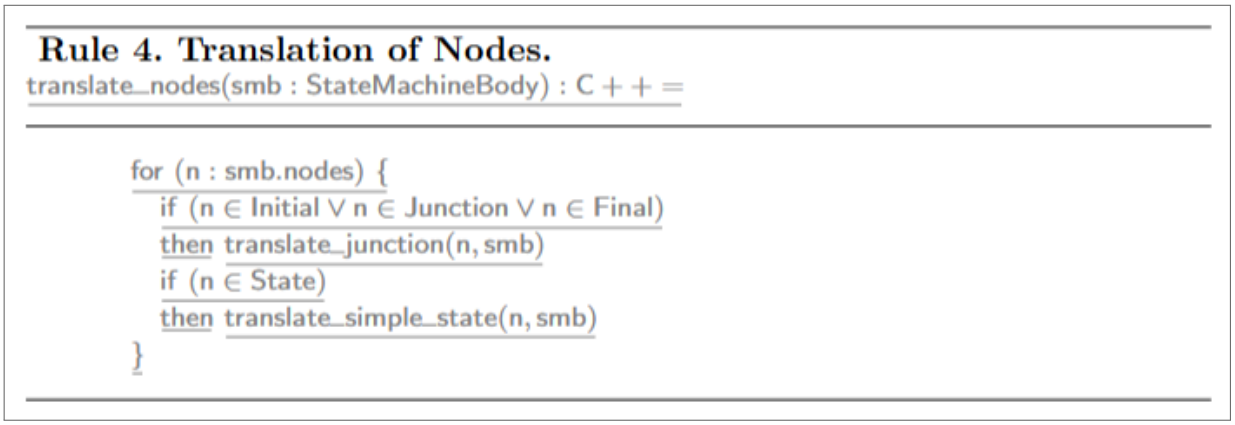
Source: Author (2025)

In our work, we consider the following node types: *Initial*, *Junction*, *Final*, and (simple) *State*. (Composite states are left for future work.) In Rule 4, we iterate over these nodes to translate them. The first three types share a common translation strategy due to their structural similarity. The node *State*, however, requires further consideration since it can perform entry and exit actions, as detailed in Rule 7. (RoboSim also supports during actions, but these are

[1] https://www.boost.org/

also left for future work.) To address the general case, we focus here on the translation of a *State*.

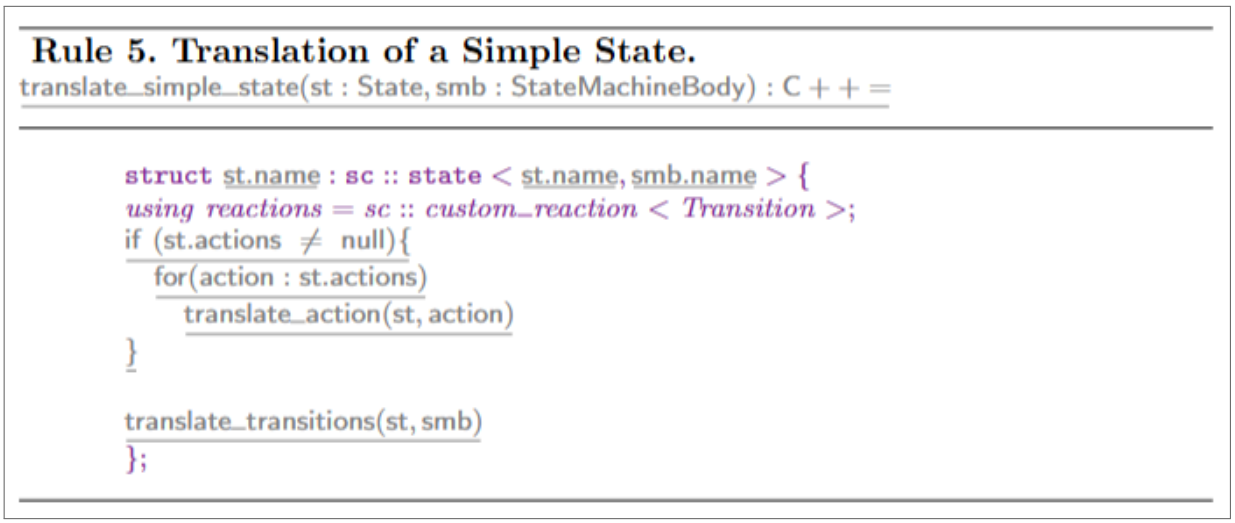
Figure 10 – Translation rule for SMB nodes



Source: Autor (2025)

In summary, each node is translated into a C++ structure that inherits the state definition provided by libboost[2] (see Rule 5). Libboost defines a state type and its characteristics, such as the context to which the state belongs. This definition requires both the node and machine names. The latter represents the context to which the node belongs. In our case, it is always the machine, since we do not consider composite states.

Figure 11 – Translation rule for a simple state



Source: Author (2025)

After creating the C++ structure, Rule 5 handles the translation of the state actions, if they exist, by invoking Rule 6. Broadly speaking, the entry action (translate_entry_action() in Rule

[2] https://www.boost.org/

7) is translated into the structure's constructor, while the exit action (translate_exit_action() in Rule 8) is translated into its destructor. In both Rules 8 and 9, we use the function translate_statement(), omitted here, to effectively translate the statements of the state action.

Finally, Rule 5 handles the translation of transitions out of the state. As specified by Rule 9, this is implemented through the *react* function, which serves as the mechanism used by Boost to manage transitions. In RoboSim, a transition has a guard condition and an action. The translation in Rule 9 begins by establishing the initial if condition, followed by a sequence of else if conditions. Each of these conditions corresponds to a transition from the translating state. Within each if (or else if) block, the transition's action is executed, followed by a call to *return transit* (a Boost function) with the target state as an argument.

Figure 12 – Translation rule for actions

Rule 6. Translation of an Action

translate_action(st : State, ac : Action) : C + + =

```
if (ac ∈ EntryAction){
   translate_entry_action(st, ac.action)
}
if (ac ∈ ExitAction){
   translate_exit_action(st, ac.action)
}
```

Source: Author (2025)

Figure 13 – Translation rule for an Entry action

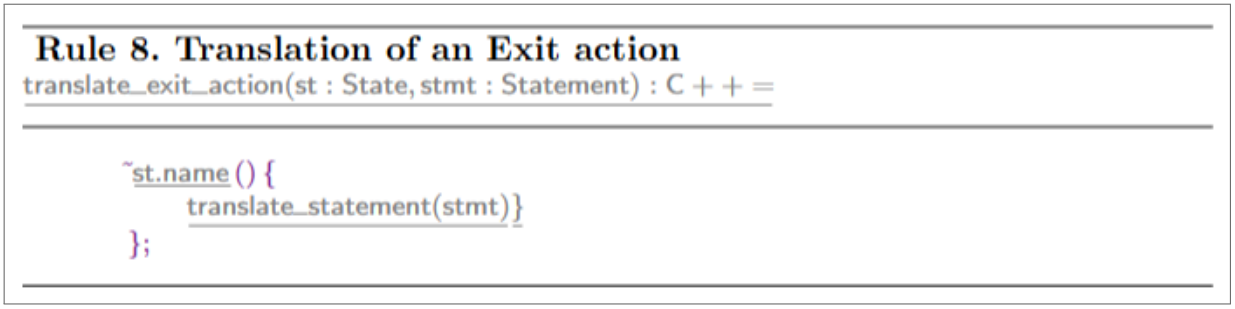
Rule 7. Translation of an Entry action

translate_entry_action(st : State, stmt : Statement) : C + + =

```
st.name(my_context ctx) : my_base(ctx){
    translate_statement(stmt)}
};
```

Source: Author (2025)

Figure 14 – Translation rule for an Exit action



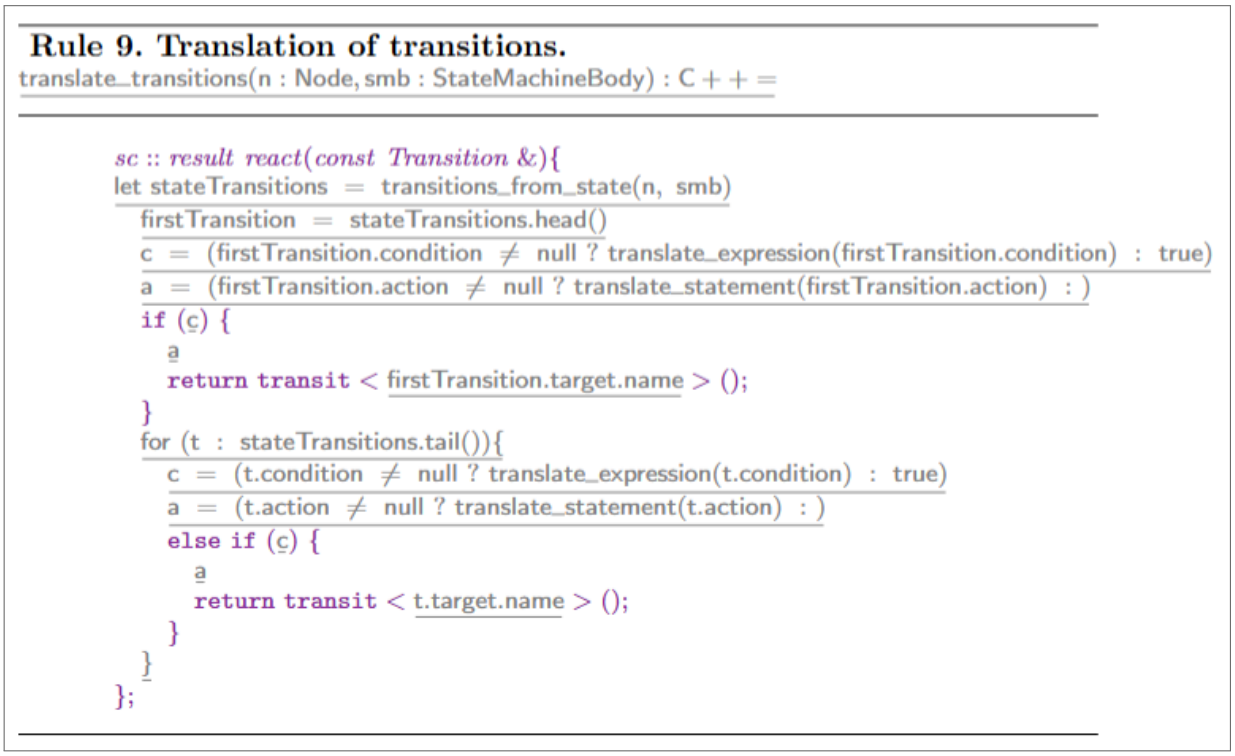
Rule 8. Translation of an Exit action

translate_exit_action(st : State, stmt : Statement) : C + + =

```
~st.name () {
    translate_statement(stmt)}
};
```

Source: Author (2025)

Figure 15 – Translation Rule for transitions



Rule 9. Translation of transitions.

translate_transitions(n : Node, smb : StateMachineBody) : C + + =

```
sc :: result react(const Transition &){
let stateTransitions = transitions_from_state(n, smb)
  firstTransition = stateTransitions.head()
  c = (firstTransition.condition ≠ null ? translate_expression(firstTransition.condition) : true)
  a = (firstTransition.action ≠ null ? translate_statement(firstTransition.action) : )
  if (c) {
    a
    return transit < firstTransition.target.name > ();
  }
  for (t : stateTransitions.tail()){
    c = (t.condition ≠ null ? translate_expression(t.condition) : true)
    a = (t.action ≠ null ? translate_statement(t.action) : )
    else if (c) {
      a
      return transit < t.target.name > ();
    }
  }
};
```

Source: Author (2025)

5 SYSTEMATIC RULE-BASED IMPLEMENTATION

To illustrate the validation of our translation rules, we have encoded a subset of the rules using the Xtend[1] framework. Xtend is a Java-based language that offers various features to handle, develop, and generate DSL code, with robust integration with Eclipse. Our implementation has been integrated with RoboTool, which is a modeling and verification tool for RoboSim.

Figure 16 – Implementation of Rule 5

```
def translateState(SimMachineDef stm, State s, HashMap<String, List<Transition>> transitionsFrom) {
    '''
    struct «qnp.getFullyQualifiedName(s).toString("_").toLowerCase()» : sc::state < «s.name», «stm.name» > {
        «IF s.actions != null»
        «FOR a : s.actions»
        «ag.translateAction(a)»

        «ENDFOR»
        «ENDIF»

        «tg.translateTransitions(s, transitionsFrom)»
    }
    '''
}
```

Source: Author (2025)

Within our strategy, every rule is implemented as an Xtend method that outputs a String, which represents the generated C++ code. As depicted in Figure 16, the implementation of Rule 5 starts by building a framework that is related to the state and then invokes qnp.getFullyQualifiedName(s) to obtain the fully qualified name of the state. Subsequently, we iterate through the state actions to translate them and call tg.translateTransitions to handle all state transitions.

Figure 17 illustrates the translation of transitions, as defined in Rule 9. This implementation starts by constructing the signature of the react function. It then extracts the condition and action from the transitions to form a sequence of "if" (and else if) statements that represent the state transitions. Consequently, upon receiving a Transition event, the react function is triggered, leading to a state change.

In RoboTool, the laws implemented are encapsulated by a plug-in called SIMCPP Generator. So, when a RoboSim model is loaded, the SIMCPP Generator can be invoked by clicking on the SIMCPP menu item in its toolbar, as shown in Figure 18. The C++ implementa-

[1] https://eclipse.dev/Xtext/xtend/

Figure 17 – Implementation of rule 9

```
def translateTransitions(State s, HashMap<String, List<Transition>> transitionsFrom) {
    var stateTransitions = transitionsFrom.get(s.name)
    var head = stateTransitions.head()
    var tail = stateTransitions.tail()
    '''
        using reactions = sc::custom_reaction<Transition>;
        sc::result react(const Transition &){
            « var c = if (head.condition !== null) exg.translateExpression(head.condition) else '''true''' »
            « var a = if (head.action !== null) stg.translateStatement(head.action) else null »
            if(«c») {
                «IF a !== null»
                «a»
                «ENDIF»
                return transit<«head.target.name»>();
            }
            «FOR t : tail»
            « c = if (t.condition !== null) exg.translateExpression(t.condition) else '''true''' »
            « a = if (t.action !== null) stg.translateStatement(t.action) else null »
            else if(«c») {
                «IF a !== null»
                «a»
                «ENDIF»
                return transit<«t.target.name»>();
            }

            «ENDFOR»
        }

    '''
}
```

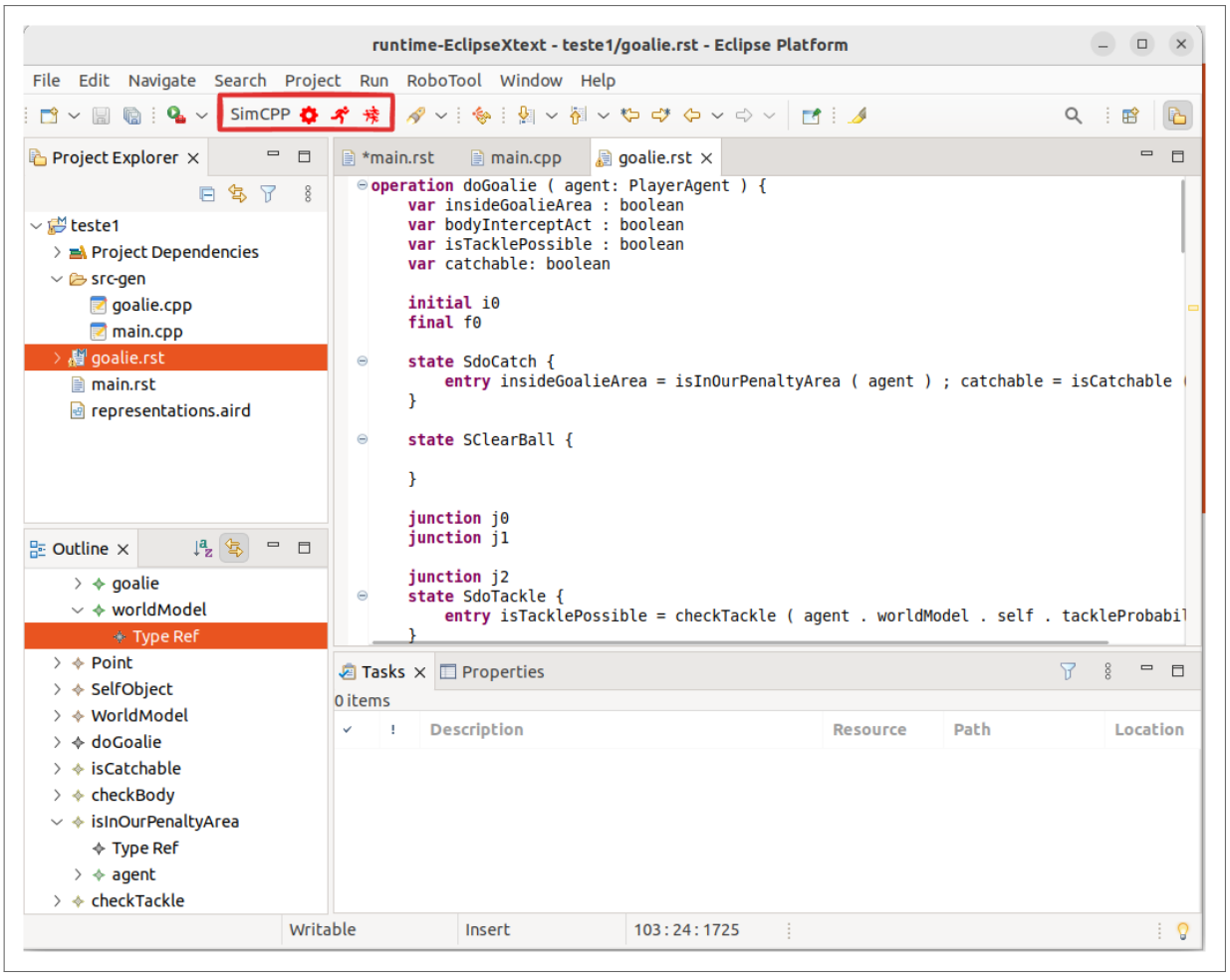
Source: Author (2025)

tion is stored in the project directory /src-gen. As an example, part of the resulting C++ implementation of the operation doGoalie() is presented in Listing 1

```
1  class DOGOALIE_Robosim {
2      struct doGoalie : sc::state_machine<doGoalie, initialStateGoalie>{};
3
4      struct dogoalie_docatch : sc::state < doCatch, doGoalie > {
5          using reactions = sc::custom_reaction<Transition>;
6          sc::result react(const Transition &){
7              if(true) return transit<f>();
8          }
9      }
10 };
```

Listing 1 – DoGoalie State Machine Implementation

Figure 18 – SimCPP puglin in RoboTool

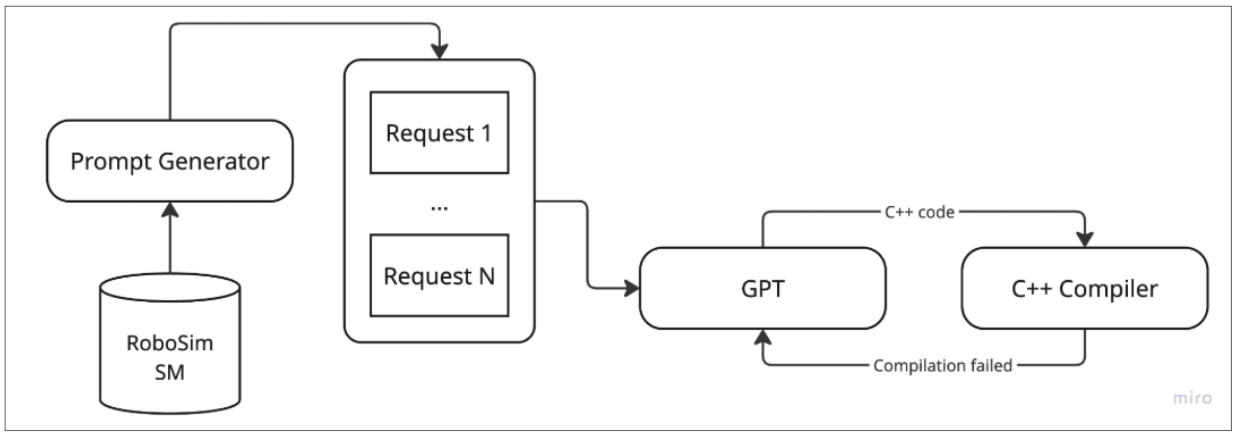


Source: Author (2025)

6 ALTERNATIVE IMPLEMENTATION STRATEGY USING LLM

In addition, large language models serve as an effective mechanism for code translation between programming languages. The established workflow, as illustrated in Figure 19, collects the RoboSim file within a prompt and asks GPT to generate the equivalent C++ file that interprets the state machine or the software operation. Subsequently, the code undergoes automatic compilation, and in the event of an error, the feedback is reintroduced through a loop strategy (LEITE et al., 2024).

Figure 19 – LLM framework architecture



Source: Author (2025)

The few-shot prompt strategy was chosen (BROWN et al., 2020) for its simplicity and effectiveness. Few-shot prompt is a prompt technique that enables in-context learning by providing demonstrations in the prompt to steer the model better performances. Our base prompt shown in Figure 20 is made up of:

- A set of rules that the LLM must follow.
- A set of examples to show what the expected result is for a RoboSim state machine.
- The translation rules presented in 4.2
- The RoboSim file for translation.

The outcome deviated from our expectations. The LLM was unable to produce functional results, as none of the generated code was capable of successful compilation. Consequently, we categorized the results to evaluate the framework's capabilities by examining the following aspects:

Figure 20 – Prompt Template

```
## Rules
- MUST CHANGE server::Vector2D to rcsc::Vector2D
- MUST CHANGE server::PlayerAgent to rcsc::PlayerAgent
- MUST use rcsc::PlayerAgent as pointer
- MUST add the conditional bodies in the transitions
- MUST include function parameters in functions
- MUST use just this imports:
    #include "rcsc/geom/vector_2d.h"
    #include <boost/statechart/custom_reaction.hpp>
    #include <boost/statechart/termination.hpp>

    #include <boost/statechart/event.hpp>
    #include <boost/statechart/result.hpp>
    #include <boost/statechart/simple_state.hpp>
    #include <boost/statechart/state.hpp>
    #include <boost/statechart/state_machine.hpp>
    #include <boost/statechart/transition.hpp>
    #include <rcsc/player/soccer_action.h>
    #include <rcsc/player/world_model.h>
    #include <rcsc/player/player_agent.h>

<EXAMPLES>
<TRANSLATION RULES>

## Requested State Machine
Here follows the RoboSim state machine about the <X> behavior, give me the equivalent C++
code, using Boost.Statechart C++ library and librcsc C++ library from RoboCup 2D.
```

Source: Author (2025)

- Import libboost.
- Imports librcsc.
- Creates all states.
- Creates all transitions.
- Implements all conditions.
- Implement actions.

The findings indicate that, to some extent, the model can understand the intended task assigned to it. The results in 1 for each criterion. We divided our tests into two categories; The first category creates requests with a prompt containing the rules from 4.2, and the second category creates requests with a prompt without rules from 4.2. A total of 80 requests were performed, 40 for each category. However, without context with respect to RoboSim, the task

proved challenging, requiring that we manually implemented the translation rules, as presented in the previous section. Most likely, this is a consequence of the fact that both RoboSim and the libraries used in the C++ implementation are not widely used. Therefore, an LLM such as GPT was probably not sufficiently trained in these notation. But, one important finding is that the LLM when translating the model implements function bodies that are not described within the model beyond the signature.

Component	**With Rules**	**No Rules**
LIBBOOST	40	40
LIBRCSC	40	40
STATES	40	40
TRANSITIONS	39	20
CONDITIONS	0	0
ACTIONS	40	40
TOTAL REQUESTS	40	40

Table 1 – Number of successful executions per component in the few-shot scenarios with and without rules.

7 CONCLUSION

This study proposes a comprehensive translation from RoboSim simulations to C++ implementations. This translation framework, based on a set of rules, generates behaviors for the RoboCup Simulation 2D environment. In summary, our strategy starts from the design and verification (of classical and domain-specific properties) of simulation models in RoboSim, yielding the automatic generation of C++ source code.

An Eclipse plug-in was developed as a practical tool that gathers a RoboSim model and translates to a compilable C++ code for the Robocup Soccer Simulation 2D. That means that it is possible to integrate the entire process of designing and implementing a soccer behavior and taking advantage of the formal verification of the RoboSim semantics in CSP.

We have also explored an LLM based translation and our findings is that this technology needs fine-tuning to better understand the RoboSim notation and the C++ libraries used in this work. This can potentially improve the GPT capabilities to make correct inferences and achieve a more consistent translation. Even so the use of LLM has shown to be promising despite imprecisions. It goes beyond translating the model; it also implements the functions and datatypes described in RoboSim models, and especially with functions from the model it tries to implement the function body based on the function signature.

In future work, our objective is to integrate Large Language Models (LLMs) to enhance model comprehension, automate the implementation of complete 2D simulation behaviors, and incorporate them directly into the generated source code. LLMs are recognized as powerful tools that consistently demonstrate strong performance in understanding and translating across diverse domains. In addition, we plan to use the fine-tuning strategy to train the LLM for our specific context. Furthermore, we plan to expand the project to operate in multiple environments and support more complex behaviors, which will require the definition and implementation of additional translation rules to cover currently unsupported RoboSim features. Furthermore, it is essential to link the generated behaviors with various effectiveness tests to assess whether they improve the performance of Robocup Simulation 2D agents.

REFERENCES

AKIYAMA, H.; NAKASHIMA, T. Helios base: An open source package for the robocup soccer 2d simulation. In: BEHNKE, S.; VELOSO, M.; VISSER, A.; XIONG, R. (Ed.). *RoboCup 2013: Robot World Cup XVII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 528–535. ISBN 978-3-662-44468-9.

BAZYDŁO, G.; ADAMSKI, M.; STEFANOWICZ, Ł. Translation uml diagrams into verilog. University of Zielona Góra, Zielona Góra, Poland, 2014.

BROOKES, S. D.; ROSCOE, A. Csp: A practical process algebra. In: _____. *Theories of Programming: The Life and Works of Tony Hoare*. 1. ed. New York, NY, USA: Association for Computing Machinery, 2021. p. 187–222. ISBN 9781450387286. Disponível em: <https://doi.org/10.1145/3477355.3477365>.

BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J.; DHARIWAL, P.; NEELAKANTAN, A.; SHYAM, P.; SASTRY, G.; ASKELL, A.; AGARWAL, S.; HERBERT-VOSS, A.; KRUEGER, G.; HENIGHAN, T.; CHILD, R.; RAMESH, A.; ZIEGLER, D. M.; WU, J.; WINTER, C.; HESSE, C.; CHEN, M.; SIGLER, E.; LITWIN, M.; GRAY, S.; CHESS, B.; CLARK, J.; BERNER, C.; MCCANDLISH, S.; RADFORD, A.; SUTSKEVER, I.; AMODEI, D. *Language Models are Few-Shot Learners*. 2020. Disponível em: <https://arxiv.org/abs/2005.14165>.

CAVALCANTI, A. L. C.; SAMPAIO, A. C. A.; MIYAZAWA, A.; RIBEIRO, P.; FILHO, M. C.; DIDIER, A.; LI, W.; TIMMIS, J. Verified simulation for robotics. *Science of Computer Programming*, v. 174, 2019. Disponível em: <papers/CSMRCD19.pdf>.

CHALLENGER, M.; DEMIRKOL, S.; GETIR, S.; MERNIK, M.; KARDAS, G.; KOSAR, T. On the use of a domain-specific modeling language in the development of multiagent systems. *Engineering Applications of Artificial Intelligence*, v. 28, p. 111–141, 2014. ISSN 0952-1976. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0952197613002297>.

CHEN, M.; DORER, K.; FOROUGHI, E.; HEINTZ, F.; HUANG, Z.; KAPETANAKIS, S.; KOSTIADIS, K.; KUMMENEJE, J.; MURRAY, J.; NODA, I.; OBST, O.; RILEY, P.; STEFFENS, T.; WANG, Y.; YIN, X. Robocup soccer server for soccer server version 7.07 and later: User manual. 2003. Disponível em: <https://rcsoccersim.github.io/rcssserver-manual-20030211.pdf>.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. W. FDR3 - A Modern Refinement Checker for CSP. In: *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2014. p. 187–201.

ISOBE, Y.; MIYAMOTO, N.; ANDO, N.; OIWA, Y. Formal modeling and verification of concurrent fsms: Case study on event-based cooperative transport robots. October 2021.

LEITE, G.; ARRUDA, F.; ANTONINO, P.; SAMPAIO, A.; ROSCOE, A. W. Extracting formal smart-contract specifications from natural language with llms. In: MARMSOLER, D.; SUN, M. (Ed.). *Formal Aspects of Component Software*. Cham: Springer Nature Switzerland, 2024. p. 109–126. ISBN 978-3-031-71261-6.

MIYAZAWA, A.; RIBEIRO, P.; LI, W.; CAVALCANTI, A.; TIMMIS, J.; WOODCOCK, J. Robochart: modelling and verification of the functional behaviour of robotic applications. *Softw. Syst. Model.*, Springer-Verlag, Berlin, Heidelberg, v. 18, n. 5, p. 3097–3149, out. 2019. ISSN 1619-1366. Disponível em: <https://doi.org/10.1007/s10270-018-00710-z>.

NODA, I.; MATSUBARA, H. Soccer server and researches on multi-agent systems. 01 1996.

PROKOPENKO, M.; WANG, P. *Gliders2d: Source Code Base for RoboCup 2D Soccer Simulation League*. 2018.

PROKOPENKO, M.; WANG, P. Gliders2d: Source code base for robocup 2d soccer simulation league. In: CHALUP, S.; NIEMUELLER, T.; SUTHAKORN, J.; WILLIAMS, M.-A. (Ed.). *RoboCup 2019: Robot World Cup XXIII*. Cham: Springer International Publishing, 2019. p. 418–428. ISBN 978-3-030-35699-6.

ROSCOE, A. W. *Understanding Concurrent Systems*. [S.l.]: Springer, 2011. (Texts in Computer Science).

SANTOS, M.; FILHO, M. C.; SAMPAIO, A. A model-based approach to the development and verification of robotic systems for competitions. In: *2023 Latin American Robotics Symposium (LARS), Brazilian Symposium on Robotics (SBR), and Workshop on Robotics in Education (WRE)*. [S.l.]: IEEE, 2023. p. 1–8.

YE, K.; FOSTER, S.; WOODCOCK, J. Formally verified animation for robochart using interaction trees. *Journal of Systems Architecture*, Elsevier, v. 140, p. 102794, 2023. ISSN 1383-7621.

ZARE, N.; AMINI, O.; SAYAREH, A.; SARVMAILI, M.; FIROUZKOUHI, A.; RAD, S. R.; MATWIN, S.; SOARES, A. *Cyrus2D base: Source Code Base for RoboCup 2D Soccer Simulation League*. 2022. Disponível em: <https://arxiv.org/abs/2211.08585>.

ZHANG, M.; DU, D.; SAMPAIO, A.; CAVALCANTI, A.; FILHO, M. C.; ZHANG, M. Transforming robosim models into uppaal. In: *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. [S.l.: s.n.], 2021. p. 79–86.

ŽLAJPAH, L. Simulation in robotics. *Mathematics and Computers in Simulation*, v. 79, n. 4, p. 879–897, 2008. ISSN 0378-4754. 5th Vienna International Conference on Mathematical Modelling/Workshop on Scientific Computing in Electronic Engineering of the 2006 International Conference on Computational Science/Structural Dynamical Systems: Computational Aspects. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0378475408001183>.

APPENDIX A – AUXILIARY TRANSLATION FUNCTIONS

Figure 21 – Translation rule for variables

Function 4. Translation of Variables.

variable_declarations(smb : StateMachineBody) : C + + =

```
for(variable : smb.variables){
  variable_declaration(variable)
}
```

Source: Author (2025)

Figure 22 – Translation rule for variable

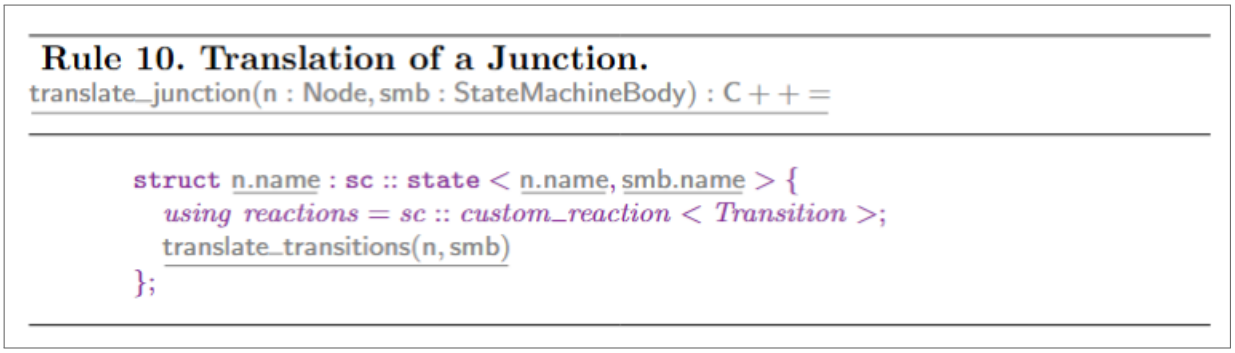
Function 5. Variable Declaration.

variable_declaration(v : Variable) : C + + =

```
if (v.modifier = var)
then {
    if (v.initial = null){
    then
      static [[v.type]]Type v.name;
    else
      static [[v.type]]Type v.name = [[v.exp]]Exp ;
}
else {
    if (v.initial = null){
    then
      static const [[v.type]]Type v.name;
    else
      static const [[v.type]]Type v.name = [[v.exp]]Exp ;
}
```

Source: Author (2025)

Figure 23 – Translation rule for an Junction



Rule 10. Translation of a Junction.

translate_junction(n : Node, smb : StateMachineBody) : C++ =

```
struct n.name : sc :: state < n.name, smb.name > {
  using reactions = sc :: custom_reaction < Transition >;
  translate_transitions(n, smb)
};
```

Source: Author (2025)