UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Lucas Francisco Pereira de Gois Correia

A Strategy to Compare Autonomous Vacuum Cleaner Algorithms Based on Coverage Path Planning in a Grid-Based Map

Recife

2025

Lucas Francisco Pereira de Gois Correia

A Strategy to Compare Autonomous Vacuum Cleaner Algorithms Based on Coverage Path Planning in a Grid-Based Map

M.Sc. Dissertation presented to the Centro de Informática of the Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Master of Computer Science.

Concentration Area: Software Engineering and Programming Languages

Supervisor: Alexandre Cabral Mota

Co-supervisor: Sidney de Carvalho Nogueira

Recife

2025

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Correia, Lucas Francisco Pereira de Gois.
A strategy to compare autonomous vacuum cleaner algorithms based on coverage path planning in a grid-based map / Lucas Francisco Pereira de Gois Correia. - Recife, 2025.
73f.: il.

Dissertação (Mestrado)- Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-Graduação em Ciência da Computação, 2025.
Orientação: Alexandre Cabral Mota.
Coorientação: Sidney de Carvalho Nogueira.

1. Robôs de limpeza autônomos; 2. Verificação de modelos; 3. Verificação formal; 4. Mapas baseados em grid; 5. Avaliação de algoritmos. I. Mota, Alexandre Cabral. II. Nogueira, Sidney de Carvalho. III. Título.

UFPE-Biblioteca Central

Lucas Francisco Pereira de Gois Correia

“Comparing Autonomous Vacuum Cleaners Based on Coverage Path Planning of Grid-based Maps with Model-Checking”

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Progamação.

Aprovado em: 21/08/2025.

ACKNOWLEDGEMENTS

I would like to first thank God for giving me the willpower and courage to make it this far. I thank my parents, Jorge Francisco Correia and Elda Pereira de Gois Correia, for all the support they have given me to reach this point. I am grateful to my wife, Jennifer Jaguaribe de Lira Correia, for being by my side since the beginning of this journey, encouraging me to see it through to the end. I sincerely thank my advisor, Alexandre Cabral Mota, for his patience and kindness in answering my questions and helping me overcome the challenges along the way. I also extend my gratitude to my co-advisor, Sidney de Carvalho Nogueira, for supporting me on this journey and allowing me to start this project.

RESUMO

A integração da verificação formal com validação empirica é crucial para garantir a robustez e eficiência de sistemas robóticos autônomos. Este trabalho apresenta uma metodologia inovadora para avaliar o desempenho de robôs de limpeza autônomos utilizando a álgebra de processos Communicating Sequential Processes (CSP) e o sistema educacional RoboMind, explorando e validando a sua combinação. O Estudo estabelece a garantia de corretude para CPP utilizando CSP, enquanto RoboMind serve como um ambiente de teste controlado para validar a navegação em diferentes ambientes. A principal contribuição deste estudo é o desenvolvimento de uma modelagem formalizada que permite uma avaliação sistemática e quantitativa de quatro algoritmos de robô de limpeza, considerando eficiência de cobertura e custo computacional. A abordagem envolve o uso de CSP como linguagem formal para definir o comportamento dos algoritmos de limpeza e do RoboMind como ferramenta de simulação para sua execução e análise. Um estudo de caso é apresentado, aplicando essa estratégia para comparar quatro algoritmos distintos de robôs de limpeza. Os resultados demonstram a eficácia da abordagem proposta, provendo uma forte garantia de corretude enquanto aprimora a aplicabilidade na simulação. Além disso, esta pesquisa destaca o potencial da combinação de CSP e RoboMind para investigações no domínio dos sistemas autônomos.

Palavras-chaves: Robôs de limpeza autônomos. Verificação de Modelos. Verificação Formal. Mapas baseados em grid. Avaliação de Algoritmos.

ABSTRACT

The integration of formal verification with empirical validation is crucial for ensuring the robustness and efficiency of autonomous robotic systems. This work introduces an innovative methodology for assessing the performance of AVCs using the process algebra Communicating Sequential Processes (CSP) and the educational system RoboMind by exploring and validating their combination. By leveraging CSP, the study establishes correctness guarantees for CPP, while RoboMind serves as a controlled testbed to validate navigation strategies in different environments. The primary contribution of this work is developing a formalized modeling enabling a systematic and quantitative evaluation of four AVC algorithms concerning coverage efficiency and computational cost. It involves employing CSP as a formal language to define the behavior of vacuum cleaner algorithms and RoboMind as a simulation tool to execute and analyze them. A case study is presented applying this strategy, comparing four different algorithms for independent vacuum cleaners. The results demonstrate the efficacy of the suggested approach, providing strong correctness guarantees while enhancing simulation applicability. Furthermore, this research underscores the potential of merging CSP and RoboMind for investigative purposes in the domain of autonomous systems.

Keywords: Autonomous vacuum cleaners; Model checking; Formal verification; Grid-based maps; Algorithm evaluation.

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

AVC	Autonomous Vacuum Cleaner
CSP	Communicating Sequential Processes
CSP_M	Machine-readable Communicating Sequential Processes
$UFPE$	Universidade Federal de Pernambuco
FDR	Failures-Divergences Refinement
CPP	Coverage Path Planning
$CPRS$	Cyber-Physical Robotic System

SUMMARY

1 INTRODUCTION

In this work, we propose an innovative strategy to compare the effectiveness of different AVC algorithms using CSP (HOARE, 1978) and the educational platform RoboMind (RoboMind.net, 2006). Below, we detail the motivation for the study, its objectives, the research questions that guide the investigation, and the structure of the work.

1.1 MOTIVATION

AVCs have become quite popular in recent years due to their ability to automate house cleaning without much human involvement (SCHNEIDERS et al., 2021). However, as various algorithms to navigate environments have been designed for these devices (RICKERT; SIEVERLING; BROCK, 2014; ESCHMANN; EBEL; EBERHARD, 2023; WANG et al., 2020), the need to precisely evaluate their performance and efficiency has increased. Precisely evaluating these algorithms is essential to understand their limitations, optimize their strategies, and minimize computational effort. This work contributes to this analysis by exploring formal methods to validate and compare AVC algorithms in structured maps. Formal methods offer a systematic and mathematically grounded approach to ensuring precise and repeatable assessments.

The impact of robot size on navigability and the coverage efficiency are crucial factors to consider. The dimensions of different robots, which require specialized analysis, can play a vital role in their performance, as well as their movement constraints and ability to avoid obstacles. Assessing verification methodologies through refinement techniques, such as a binary search-based algorithm, provided insights into the algorithm's efficiency, allowing us to select the highest coverage under time constraints.

RoboMind is an introductory environment to programming and robotics designed for educational purposes. It uses the ROBO programming language specialized in robotics. Such a language can mimic robot sensors like color reading and obstacle detection. Moreover, the virtual robot can interact with the environment, using commands that grab objects and paint the floor.

The primary reason for selecting RoboMind over other simulators, such as Gazebo or ROS, was that most of these simulators focus on real-time robotics applications with complex

Repository with the code used in this work can be found here: <https://github.com/LucasFranciscoCorreia/Comparing-Autonomous-Vacuum-Cleaners>

physics-based simulations instead of providing a determinist execution environment, ensuring reproducibility in algorithm analysis. Reproducibility is critical for a structured and predictable framework to validate CPP algorithms mathematically. Model-checking in deterministic environments ensures that a given control strategy always behaves as expected under specific conditions, which is necessary before introducing probabilistic variations. Additionally, RoboMind can complement stochastic models in a hybrid approach by verifying logical correctness and integrating stochastic simulations afterward to test robustness against variations.

We chose the ROBO language (from RoboMind (RoboMind.net, 2006)) because it fits our formalization and verification goals:

- Deterministic, discrete semantics on a grid world, which matches our abstraction for CPP and enables reproducible experiments.
- Small, well-defined instruction set (move, turn, sense, paint, pick/drop), allowing a direct and automated translation to CSP events and processes.
- Explicit, observable actions and sensor queries, which simplify the definition of alphabets and refinement checks in FDR.
- Lightweight tooling and fast execution, making large batches of runs feasible without the overhead of physics engines.

Many studies on AVCs that utilize CPP for grid-based maps often depend on simulations or empirical testing as their main verification methods (GOVINDARAJU et al., 2023). CSP provides a rigorous mathematical framework for modeling concurrency through message-passing semantics (HOARE, 1978). Autonomous robotic systems are increasingly complex and operate in critical environments, where failure can lead to serious consequences. To ensure the safety and correctness of such systems, formal methods provide a rigorous mathematical foundation for design and verification. Among these, CSP stands out as an effective approach for modeling concurrent behaviors, detecting design flaws early, and ensuring the satisfaction of critical system properties throughout the development cycle. The grid-based model serves as an abstraction of the real world, with the treatment of associated uncertainties deferred to Chapter 6.

Formal verification techniques have emerged to complement empirical validation by enabling exhaustive analysis of robotic control systems. Techniques such as model checking and theorem proving allow developers to ensure critical properties like deadlock freedom and termination correctness before physical testing begins (LUCKCUCK; FARRELL; FISHER, 2021). These techniques are particularly valuable in autonomous robotics, where systems operate under uncertainty and demand high reliability (MECK; WAGNER, 2018).

Formal methods are mathematically based approaches that facilitate the assertion and reasoning about system behavior, which is relevant in software design and engineering. Using formal methods for autonomous systems, such as AVCs, expresses the importance of the proper design and development processes while highlighting an accurate and precise description.

For CPRSs, formal methods have been applied to integrate formal techniques throughout the software development lifecycle providing precise techniques for specifying, designing, and verifying software systems, making them more accessible and applicable to real-world CPRS projects (GHASSEMI; TRIPAKIS, 2020). Within this context, CSP, Petri Nets, and Timed Automata all have precise, well-established semantics. In this work we favor CSP because (i) its explicit event alphabets and parallel composition make assume-guarantee and modular refinement checks convenient; (ii) the FDR tool provides automated refinement checking (traces/failures/divergences) and counterexample traces that we exploit to derive coverage paths (ROSCOE, 1995; GIBSON-ROBINSON et al., 2014); and (iii) our properties are untimed, so time/clock structures of Timed Automata add little benefit here. Petri nets and Timed Automata remain viable alternatives—particularly when state-space structure or quantitative timing is central—but our analysis benefits from CSP's event-centric viewpoint and refinement-oriented tooling.

The proposed strategy involves applying CSP as a formal method for defining the semantics of AVC algorithms and using RoboMind as a simulation tool to execute and test them.

CSP provides a rigorous methodology to specify the behavior of concurrent and sequential systems, while RoboMind simulates the actions of robots in virtual environments. The approach involves modeling and verifying robots in a grid-based map, where we measure how much of the map the robot can clean.

Moreover, this work evaluates deterministic and non-deterministic path-planning algorithms, highlighting their advantages and trade-offs. This comparison is essential to understand the efficiency of different strategies in achieving optimal coverage with minimal effort and computational cost.

1.2 OBJECTIVES

The main objective of this work is to develop a formal strategy combining modeling and verification strategy, allowing a quantitative assessment concerning coverage efficiency and computational cost to compare the effectiveness of distinct AVC algorithms. To achieve the main objective, we aim to:

- Specify quantitative metrics for coverage and computational effort, and the properties to be checked (e.g., deadlock freedom, termination);
- Implement an automated pipeline (RoboMind execution + FDR refinement checking) to generate and analyze traces;
- Compare representative algorithms (Random Bounce, Improved Random Bounce, DFS, Inwards Spiral) across selected maps;
- Analyze the relationship between computational effort and coverage of the algorithms;
- Evaluate the impact of robot size on path feasibility and achievable coverage; and
- Apply a binary-search-based refinement procedure to determine the maximum coverage under time constraints.

We adopt CSP and RoboMind as the underlying modeling and execution framework to realize these goals.

1.3 RESEARCH QUESTIONS

To guide our investigation, we formulate the following research questions:

- How can AVC algorithms be formalized using CSP?
- What is the relationship between coverage efficiency and computational cost among different algorithms?
- How can the combination of CSP and RoboMind contribute to evaluating autonomous robotic algorithms?

- Does the proposed approach allow the identification of patterns and limitations in autonomous cleaning algorithms?
- How does robot size impact cleaning efficiency and verification complexity?

1.4 CONTRIBUTIONS

By combining CSP and RoboMind, we propose an assessment environment that can accurately model the trajectories of AVCs and allow us to gauge their productivity and performance in different situations. In contrast, our approach employs model-checking with CSP, providing a rigorous means of verifying correctness that simulation techniques do not always guaran tee. This work uses formal verification to identify and eliminate logical inconsistencies before deploying the system in real-world environments.

To demonstrate the effectiveness of our approach, we present a case study comparing four distinct algorithms for AVCs (Code_1 (Random Bounce), Code_2 (Improved Random Bounce), Depth-First Search and Inwards Spiral). The results demonstrate the efficacy of our approach in providing insights into the strengths and weaknesses of different algorithms. Furthermore, this study highlights the potential of combining CSP and RoboMind for research purposes. Overall, this work provides a formal and objective approach to evaluating the performance of AVCs, which can be used by researchers, manufacturers, and consumers alike to make informed decisions about these devices.

The main contributions of this work are:

- A strategy to evaluate AVCs algorithms using a formal language in a well-defined grid-based environment;
- Extract counterexample traces which represent an AVC algorithm in terms of CSP concerning the best trajectory for covering path planning using grid-based maps;
- A methodology that incorporates robot size constraints into coverage evaluation;
- The application of binary search-based refinement techniques to enhance coverage verification accuracy within time constraints.

1.5 STRUCTURE OF THE WORK

The remainder of this work is organized as follows:

- Chapter 2 presents the background concepts, including the RoboMind platform and the process algebra CSP.
- Chapter 3 introduces our validation strategy, covering the map selection process, its formal representation, and using CSP for algorithm validation.
- Chapter 4 discusses the validation of the proposed approach, presenting examples of ROBO code used for CSP specifications and corresponding results.
- Chapter 5 reviews related work on path planning for robotic applications.
- Chapter 6 outlines the internal and external threats that could affect the approach's validity.
- Finally, Chapter 7 summarizes the main conclusions and suggests directions for future work.

2 BACKGROUND

This section provides the main concepts of RoboMind and the process algebra CSP.

2.1 ROBOMIND

RoboMind is a simulation and educational environment to analyse and design robot systems. It provides a programmable virtual robot moving in a 2D grid world and a simple domain-specific language (ROBO) for specifying its behaviour. It is aimed at teaching programming and computational thinking while allowing users to experiment with basic robotics concepts. RoboMind provides the capabilities to:

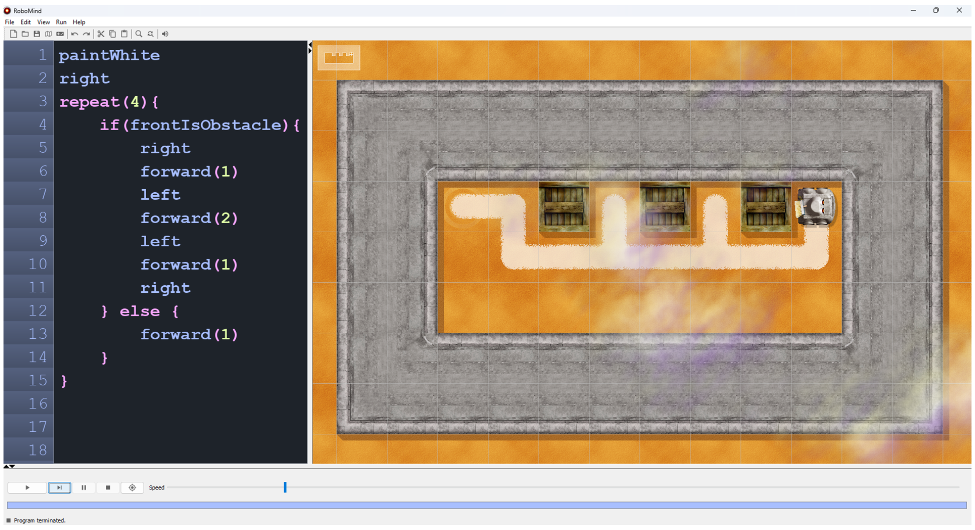
- define grid-based maps using a textual or graphical representation, with walls, obstacles and free cells;
- program the virtual robot in the ROBO language using commands for movement, turning, painting, sensing and control flow;
- execute and visualize robot programs step by step or at different speeds, observing the resulting trajectory and painted cells;
- simulate simple sensing, such as obstacle and colour detection, in a deterministic and reproducible way;
- load, save and share maps and programs for experimentation and teaching.

Selecting an appropriate simulation environment is crucial for evaluating AVC algorithms. The RoboMind environment was chosen for this research due to its modeling capabilities for the main aspects of the environment relevant to cleaning algorithms.

In Figure 1, we show the RoboMind environment, with the robot simulation on the right side and the ROBO language used to control it on the left. In the robot simulation, the robot walks east from the top-left corner of the map until reaching the top-right corner. The bottom section of the figure shows simulation controls, including start, stop, and speed adjustment.

The RoboMind environment enables programming and simulating a virtual robot in a map for learning programming logic and developing computational thinking by writing code in the ROBO language. ROBO is a simple educational programming language that is developed to

Figure 1 – Illustration of the RoboMind environment
Source: https://robomind.net/



```
paintWhite
right
repeat(4){
    if(frontIsObstacle){
        right
        forward(1)
        left
        forward(2)
        left
        forward(1)
        right
    } else {
        forward(1)
    }
}
```

allow the user to program immediately, using a concise set of commands designed to control a robot. RoboMind allows one to visualize and interactively simulate the robot's actions. We show the main aspects of the ROBO language in the Table 1.

A map in the RoboMind environment is defined using a textual representation to allow the graphical representation shown in Figure 1. Each capitalized letter denotes a visual element of a wall (e.g., "C" represents the top-left corner), the "@" symbol indicates the robot's starting position and the free spaces represent open areas where the robot can move freely.

```
map: CHHHHHHHHHHD
     GMFFFFFFFFJI
     GI@ Q Q Q GI
     GI        GI
     GI        GI
     GLHHHHHHHHKI
     BFFFFFFFFFFE
```

Each cell in the textual map corresponds to a coordinate (i,j), allowing a one-to-one mapping of spatial elements into formal variables.

While RoboMind provides a simplified environment compared to real-world AVCs, its structured and deterministic nature enables precise verification of control strategies without the noise introduced by real-world uncertainties by focusing on the logical correctness of robot behavior rather than on low-level physics. The primary goal is not to simulate real-world physics in full detail but rather to establish a rigorous correctness proof for core behaviors such as navigation, obstacle avoidance, and task completion. This approach follows the well-established principle in formal verification that logical soundness should be established first before addressing physical complexities through additional simulation or empirical validation.

Table 1 – Main elements of the ROBO language

ROBO	**Semantics**
`forward(n)`	Moves n steps forward
`left()`	Turns left over 90º degrees
`right()`	Turns right over 90º degrees
`north(n)`	Turns north and move n steps forward
`paintWhite()`	Starts painting the ground with a white paint
`stopPainting()`	Stops painting
`flipCoin()`	Flips a coin to make a random choice. flipCoin() will either be true or false with a chance of $50\% - 50\%$
`frontIsObstacle()`	Verifies if the square in front of the robot contains an obstacle. Returns true if it contains an obstacle, otherwise it returns false
`frontIsClear()`	Verifies if the square in front of the robot does not contains an obstacle. Returns true if its clear, otherwise it returns false
`frontIsWhite()`	Verifies if the square in front of the robot is painted with a white paint. Returns true if its painted white, otherwise it returns false
`repeat(n) { ... }`	Repeats a set of commands n times
`repeatWhile( ... ) { ... }`	Repeats a set of commands while its condition is true
`if ( ... ) { ... } else { ... }`	Executes a set of commands if its condition is true. Otherwise, it executes another set of commands
`procedure <name> ( ... ) { ... }`	Creates a procedure to be called by the language. It receives a set of commands to execute and can be called by executing <name>() with its parameter values, if it has any.

While simulators like Gazebo (FOUNDATION, 2024) or ROS (ROBOTICS, 2024) provide realistic simulations, they also introduce higher computational costs and complexity, making formal verification challenging. In (GHASSEMI; TRIPAKIS, 2020), when discussing complexity trade-offs, they also come with significant computational overhead and increased debugging complexity. The objective of using RoboMind in our approach is not to replace these simulators but to complement them by leveraging formal verification techniques early in the development process. Unlike purely physics-based simulators, CSP-based verification allows us to prove properties such as coverage guarantees, deadlock absence, and correct decision-making in an abstract yet rigorous manner. RoboMind, by contrast, excels in scenarios where formal validation of decision-making processes is paramount. Rather than replacing Gazebo or ROS, RoboMind should be considered a complementary tool that enables rigorous verification before deploying models in high-fidelity simulators or real-world environments. This hybrid approach helps ensure that control algorithms are effective and provably correct before being subjected to real-world uncertainties.

CPP is an important activity in walking robots, primarily used in AVCs. RoboMind offers a convenient way of modeling grid-based maps, which is significant for this activity. Thus, the platform's features are suitable for simulating multiple vacuum cleaning algorithms. While an abstraction, the grid-based model captures AVCs' essential discrete decision-making processes. While grid-based maps are a simplified model, they do not inherently imply poor real-world applicability; instead, they enable us to focus on algorithmic correctness and logical guarantees before dealing with environmental noise.

This choice aligns with previous research advocating the use of deterministic and lightweight simulation platforms for formal verification. Approaches such as CSP2Turtle (DOE; SMITH, 2022) and RoboChart (CAVALCANTI, 2018) similarly bridge simulation and model checking by translating robot behavior into formal models for exhaustive analysis.

2.2 THE PROCESS ALGEBRA CSP

Formal methods have been applied across a wide range of robotic systems, from educational robots to industrial applications. Studies show their effectiveness in verifying reactive behaviors, coordinating swarm robotics, and specifying embedded architectures (CAVALCANTI, 2018; BRUNSKILL et al., 2021). Although challenges remain, such as scalability and the need for abstraction, these techniques help eliminate design flaws early, reducing deployment risk (LUCK-

CUCK; FARRELL; FISHER, 2021; MECK; WAGNER, 2018). Frameworks such as LUNA demonstrate how CSP can support real-time, embedded robotics with concurrency constraints (WILTERDINK, 2011).

CSP is a process algebra used in analyzing and verifying concurrent systems. A CSP process communicates with the environment using events. The patterns of communications are defined using CSP operators. It allows the modeling and analysis of systems where multiple processes interact through communication (ROSCOE, 1995). CSP enables the rigorous specification of the behavior of autonomous robots in a navigation environment, ensuring that desired properties are systematically verified.

CSP offers a well-defined mathematical model, allowing the automatic verification of properties such as safety, liveness, and deadlocks. Additionally, CSP has well-established tools such as Failures-Divergence Refinement (FDR) (GIBSON-ROBINSON et al., 2014), which enables refinement verification between processes and detects potential violations.

FDR, the main model-checking tool for checking refinements in CSP, inputs a dialect of CSP called CSP_M (machine-readable CSP), which embeds a functional language to express data structures, and a behavioral language to declare and compose the CSP processes. It is widely used in robotic applications due to its ability to verify deadlock-freedom, liveliness, and trace-based refinement (CAVALCANTI, 2018)

The primary reason for selecting CSP is the existence of a structured mapping from ROBO language to CSP, allowing for direct translation and verification (CORREIA, 2021). The choice of CSP in this work follows comparative studies showing its advantages for reasoning about communication in concurrent robotic systems. This choice ensures that CSP-based models align with programming practices and can be efficiently analyzed.

While formal verification in CSP can guarantee logical correctness, it does not directly account for all real-world implementation constraints (e.g., sensor noise, timing variations, hardware failures). However, formal methods are not meant to replace real-world testing; they are a crucial early step in eliminating fundamental design flaws before empirical testing begins. While empirical testing can expose issues in real-world conditions, it does not guarantee exhaustive correctness — there is always the risk of missing edge cases. Conversely, CSP provides a systematic way to verify critical properties under all possible execution paths. CSP's role is not to replace empirical testing but to complement it by ensuring correctness at an abstract level before deployment.

Integrating CSP with simulation-based approaches introduces complexity to the problem, but this complexity can be justified depending on the goals of the analysis. Rather than fully merging CSP with a simulation framework, a structured approach would use CSP before simulating to eliminate logical errors and verify key correctness properties, serving as an early correctness filter, eliminating logical errors before moving to empirical validation, ensuring that the formally verified algorithms function as expected in dynamic conditions, and reducing the likelihood of logical flaws propagating into real-world tests by providing counterexample traces, making it easier to detect issues before testing in RoboMind. While integration requires effort, the benefits — such as ensuring correctness before empirical testing — can outweigh the costs when applied strategically.

CSP stands out not only for its expressiveness in modeling concurrency, but also for enabling modular and hierarchical system composition (HOARE, 1978) — properties essential to scalable formal verification in robotics.

2.2.1 Language Elements

Table 2 summarizes the main CSP$_M$ constructs used in this work (Formal Systems (Europe) Ltd., 2024). These elements form the basis of the formalization of ROBO programs into CSP processes, enabling structured and modular modeling.

2.2.2 Refinement and Model Checking

CSP comprises three primary semantic models (ROSCOE, 2010), which record significantly different views of the processes. Specifically, in this research, we are interested in the *traces* model that records the possible sequences of events (or traces) that a process can communicate. Such a model is much less computationally intense than other semantic models and suitable for robot algorithms' verification (FILHO et al., 2018; ARAÚJO; MOTA; NOGUEIRA, 2019).

In this work, we focus on the traces model of CSP. Intuitively, a trace is a finite sequence of observable events that records one possible execution of a process. Formally, given an event alphabet Σ, a trace is a sequence $t \in \Sigma^*$, for example:

$$t = \langle move, paint, move, turnLeft \rangle.$$

For a CSP process P, $traces(P)$ denotes the set of all finite traces that P can perform.

Table 2 – Main elements of CSP$_M$ used in this work

Element	Description
`STOP`	Represents a process that does nothing
`SKIP`	Represents successful termination
`a -> P`	Performs event a and then behaves as process `P`
`datatype A = B | C | D | ...`	Turns `A` into a `data` declaration containing the set `{B, C, D, ...}`
`datatype A = B.X`	Turns `A` into a `data` declaration containing the set `{B.X}`, where `X` is a set or another `data` declaration as parameter
`M = (| k => v |)`	Turns `M` into a `Map` structure, where `M(k)` takes `O`$(\log n)$ to search `k` and return `v`
`P = let ... within ...`	Allows definitions created inside `let` to be usable only inside `within`
`P [| A |] Q`	Parallel composition synchronizing on set of events `A`
`diff(Events, A)`	Subtracts from the set of all possible events the set A
`P \ A`	Hides events from set of events `A` from external observation
`P [T= Q`	Refinement assertion where the traces of `Q` contains the traces of `P`

When FDR checks a refinement $S \sqsubseteq_T P$ in the traces model, it verifies that every trace of the implementation P is also a trace of the specification S, i.e.,

$$traces(P) \subseteq traces(S).$$

If the refinement fails, FDR returns a counterexample trace, that is, a sequence of events that is possible in P but not allowed by S.

In our context, traces correspond to sequences of robot actions (e.g., moves, turns, paint operations) observed during a run of a ROBO program on a map. We use these traces as the only behavioural model to:

- measure **effort**, by counting events (steps) in a trace;
- derive **coverage**, by mapping visited/painted cells along the trace;
- detect violations, when FDR returns counterexample traces for a given property.

Other CSP semantic models (such as failures or failures–divergences) capture refusal sets and divergence, but in this work we restrict ourselves to traces because they are sufficient for our coverage and effort analysis.

FDR generates counterexample traces if a refinement verification does not hold. A counterexample is a trace that belongs to the traces of `Q` but not to `P`. Counterexamples can give valuable information for correcting and improving robot algorithms expressed as CSP processes.

For clarity, we reproduce the ROBO program shown in Figure 1, which will be used in the demonstration below.

```
right
repeat (4) {
  if (frontIsObstacle) {
    right
    forward(1)
    left
    forward(2)
    left
    forward(1)
    right
  } else {
    forward(1)
  }
}
```

In the following CSP specification, the constant `INIT` is a map data structure that records the variables in the initial state of the program, such as the position and the robot's orientation. The map contains a name as the key and its respective value. We explain the values in the map: the first key holds the `X` coordinate of the robot, the second has the `Y` coordinate, and the third is the current direction the robot.

```
INIT = (|
  X => startX,
  Y => startY,
  ORIENTATION => NORTH
|)
```

In what follows, we show a fragment of the CSP process that captures the semantics for the ROBO program presented in Figure 1. The process `PROGRAM` behaves initially as the process `RIGHT1` whose parameter is the constant `INIT`. The behavior of this last process is defined by the process `RIGHT`. The CSP specification is defined as a structured chain of process composition. Initially, the process `RIGHT` executes, and, once completed, it behaves as `WHILE2`, which specifies the command `repeat(4)` in ROBO. The process `RIGHT` employs a let/within construct to locally define the identifier o that stores the robot's current orientation. Such a process behaves as the process next that receives the map updated with the orientation set to the right position ((o+1)%4). This chain of processes ensures that the iterative behavior in ROBO is faithfully replicated in CSP.

```
PROGRAM = RIGHT1(INIT)
RIGHT1(m) = RIGHT(m, WHILE2)
RIGHT(m, next) =
  let
    o = get(m, ORIENTATION)
  within
    next(setVar(m,ORIENTATION, (o+1)%4))
WHILE2(m) = REPEAT2(4, m, TERMINATE)
```

The process REPEAT2 specifies the behavior of the WHILE2 process. It is defined by pattern-matching in two cases: (i) When the parameter n is zero, it behaves as the process TERMINATE, which behaves as a successful termination (SKIP), and, (ii) When the parameter n is not zero, it behaves as the IF3 process, which is specified by the RIGHT4 process if in front of the robot there is an obstacle or by the FORWARD11 process if the robot's path is clear. This last process is followed by a NEXT2, specified by a REPEAT2 process with the parameter n decremented by 1. As a consequence of this specification, the process REPEAT(m, 4) behaves as the repetition of the IF3 process four times.

```
REPEAT2(0, m, next) = next(m)
REPEAT2(n, m, next) =
  let
    IF3(m) =
      let
        x = get(m, X)
        y = get(m, Y)
        o = get(m, ORIENTATION)

        RIGHT4(m) = RIGHT(m, FORWARD5)
        FORWARD5(m) = FORWARD(1, m, LEFT6)
        LEFT6(m) = LEFT(m, FORWARD7)
        FORWARD7(m) = FORWARD(2, m, LEFT8)
        LEFT8(m) = LEFT(m, FORWARD9)
        FORWARD9(m) = FORWARD(1, m, RIGHT10)
        RIGHT10(m) = RIGHT(m, NEXT2)

        FORWARD11(m) = FORWARD(1, m, NEXT2)
      within
        if(frontIsClear(x, y, o)) then (
          RIGHT4(m)
        ) else (
          FORWARD11(m)
        )
    NEXT2(m) = REPEAT2(m, n-1)
  within
    IF3(m)

TERMINATE(m) = SKIP
```

The process FORWARD is also defined by pattern-matching: (i) When the parameter n is zero, it behaves as the next process, and (ii) when the parameter n is not zero, it updates the robot's position according to the current orientation, provided the robot's path is clear. For instance, if the actual orientation is to the east, the robot will move forward in the X-axis (increment its X position). The updates for the other directions are performed similarly, considering the current direction. If an obstacle blocks the robot's path, it behaves as FORWARD(0,m,next). Considering the state of the robot simulation depicted in Figure 1, the robot starts facing north and then turns right; consequently, the next state is to face east.

```
FORWARD(0,m,next) = next(m)
FORWARD(n,m,next) =
  let
    x = get(m, X)
    y = get(m, Y)
    o = get(m, ORIENTATION)
  within
    if(frontIsClear(x,y,o)) then(
      if(o == NORTH) then (
        FORWARD(n-1, setVar(m,Y,y-1), next)
      ) else if(o == EAST) then (
        FORWARD(n-1, setVar(m,X,x+1), next)
      ) else if(o == SOUTH) then (
        FORWARD(n-1, setVar(m,Y,y+1), next)
      ) else (
        FORWARD(n-1, setVar(m,X,x-1), next)
      )
   ) else (
     FORWARD(0, m, next)
   )
```

3 STRATEGY OVERVIEW

This chapter presents the strategy proposed in this dissertation. It builds on a prior research (CORREIA, 2021), which we briefly summarize for completeness, and extends it with the following contributions:

- Memory representation of the RoboMind environment for coverage accounting;
- Formal definition of coverage and computational effort used in our evaluation;
- Analysis of robot size and path feasibility;
- Determination of maximum possible coverage under time constraints via a binary-search-based verification method.

The proposed strategy enables the assessment of cleaning algorithms represented in the ROBO language. The input for this strategy is a ROBO program and a 2D map in the RoboMind environment that is translated into a CSP specification (see Figure 2). The CSP specification is the input for the refinement checker tool FDR that verifies assertions that yield counterexample traces that exhibit the effort and coverage of a given ROBO code to cover the given 2D grid-based map.

While RoboMind provides an effective simulation environment for visualizing and testing AVC algorithms, it lacks the formal verification capabilities that rigorously validate algorithmic correctness. CSP enables exhaustive verification, ensuring the algorithm follows its intended specifications under all possible conditions. Unlike RoboMind, which relies on empirical observation, CSP provides a mathematically grounded approach that can prove algorithmic properties, such as correctness, completeness, and optimality, rather than merely testing them in specific cases.

By leveraging CSP for formal verification and RoboMind for simulation-based visualization, this research adopts a methodologically sound approach that combines the strengths of both platforms — ensuring that algorithms are not only empirically tested but also mathematically proven for correctness, efficiency, and robustness. This combination avoids redundant complexity rather than overcomplicating verification. If discrepancies arise between CSP verification and RoboMind simulations, they highlight areas where additional refinement or model adjustments are needed — this feedback loop strengthens both processes.

The structured integration of these methods provides a more explicit, more reliable verification process. Without formal verification, simulation-based approaches risk missing fundamental logical errors that only become evident later, requiring costly debugging. Rather than complicate the process, CSP acts as a first-pass filter, allowing RoboMind simulations to focus on real-world execution challenges rather than basic logic errors, ensuring that only well-defined processes move forward to empirical validation.

3.1 PREVIOUS RESEARCH

As in prior research (CORREIA, 2021), we have designed a translator that maps the syntax of a ROBO code to a CSP process with equivalent semantic meaning, used for FDR model checking. We used this translator to convert various ROBO programs into their equivalent CSP$_M$ specifications. To ensure semantic equivalence between each ROBO command and its CSP specification, we map each command to a corresponding CSP process, preserving the order of execution and decision-making logic. The CSP model allows exhaustive verification through FDR, ensuring that the translated behavior adheres to the expected properties of the original RoboMind program.

Figure 2 shows a visual representation of this prior research. Spoofax (KATS; VISSER, 2010) was the primary framework to build this translator since it supports domain-specific language creation. In this case, `SDF3` was incorporated to capture the grammar of ROBO language as a parsed syntax. Stratego (KALLEBERG, 2006) was used to manipulate the parsed syntax, guaranteeing transformation into the semantically equivalent CSP$_M$ process. This framework creates an archive that can be used in Java to translate the ROBO code into an equally semantic version in CSP. We also used Java to create a simple algorithm to translate any RoboMind map into CSP. Combining the translation and map, we can assess them with FDR to generate the required counterexamples for the assessment.

In Table 3, we demonstrate the translation process for the code presented in Figure 1. It provides a detailed view of how ROBO language commands are translated into CSP$_M$. This translation is essential to ensure the robot's execution can be formally verified in CSP$_M$. More details about the translation process for other elements from the ROBO language to CSP$_M$ can be found in (CORREIA, 2021)

Basic commands, such as `right` and `forward(n)`, have a direct correspondence in CSP$_M$ (`RIGHT(m, next)`, `FORWARD(n, m, next)`, etc.). Condition structures, such as `if-else`, are

Table 3 – Translation of a ROBO program into a semantically equivalent specification in CSP$_M$

ROBO	**CSP$_M$**

`right`	`PROGRAM = RIGHT1(INIT)`
	`RIGHT1(m) = RIGHT(m, WHILE2)`

`repeat (4) {`	`WHILE2(m) = REPEAT2(4, m, TERMINATE)`
	`REPEAT2(0, m, next) = next(m)`
	`REPEAT2(n, m, next) =`
	` let`

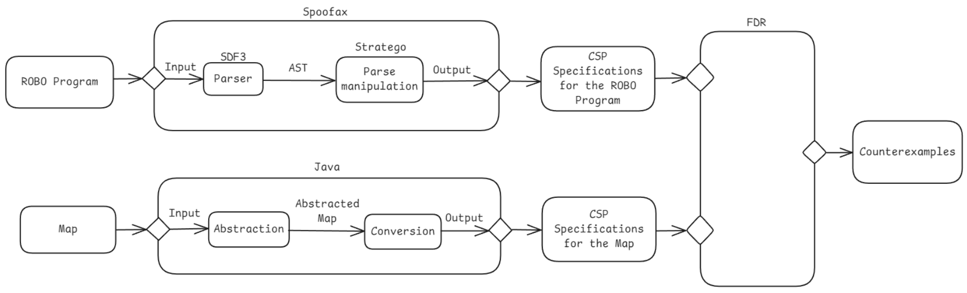
` if (frontIsObstacle) {`	`  IF3(m) =`
	`    let`
	`     x = get(m, X)`
	`     y = get(m, Y)`
	`     o = get(m, ORIENTATION)`

`  right`	`     RIGHT4(m) = RIGHT(m, FORWARD5)`
`  forward(1)`	`     FORWARD5(m) = FORWARD(1, m, LEFT6)`
`  left`	`     LEFT6(m) = RIGHT(m, FORWARD7)`
`  forward(2)`	`     FORWARD7(m) = FORWARD(2, m, LEFT8)`
`  left`	`     LEFT8(m) = LEFT(m, FORWARD9)`
`  forward(1)`	`     FORWARD9(m) = FORWARD(1, m, RIGHT10)`
`  right`	`     RIGHT10(m) = RIGHT(m, NEXT2)`
` } else {`	
`  forward(1)`	`     FORWARD11(m) = FORWARD(1, m, NEXT2)`

` }`	`    within`
	`     if(frontIsObstacle(x, y, o) then (`
	`      RIGHT4(m)`
	`     ) else (`
	`      FORWARD11(m)`
	`     )`

`}`	`  NEXT2(m) = REPEAT2(n-1, m, next)`
	` within`
	`  IF3(m)`

Figure 2 – Assessment strategy to convert ROBO to CSP and assess with FDR
Source: Author (2025)



translated into processes that evaluate conditions and direct execution to the corresponding state. Loop structures like `repeat` are modeled recursively using the corresponding CSP$_M$ process. This correspondence is fundamental for understanding how a program written in ROBO can be formally specified and analyzed as CSP$_M$.

Figures 3 and 4 complement Table 3 by providing a graphical visualization of command execution. It helps illustrate the sequence of operations within a ROBO program, the decision-making processes, especially in conditional and loop structures, and the state relationships, showing how transitions occur. Figure 3 represents the execution flow in the original ROBO program presented in Figure 1. Figure 4 shows the corresponding flow in CSP$_M$.

Both flowcharts demonstrate how the translation preserves the program's original logic.

Figure 3 – ROBO flowchart
Source: Author (2025)

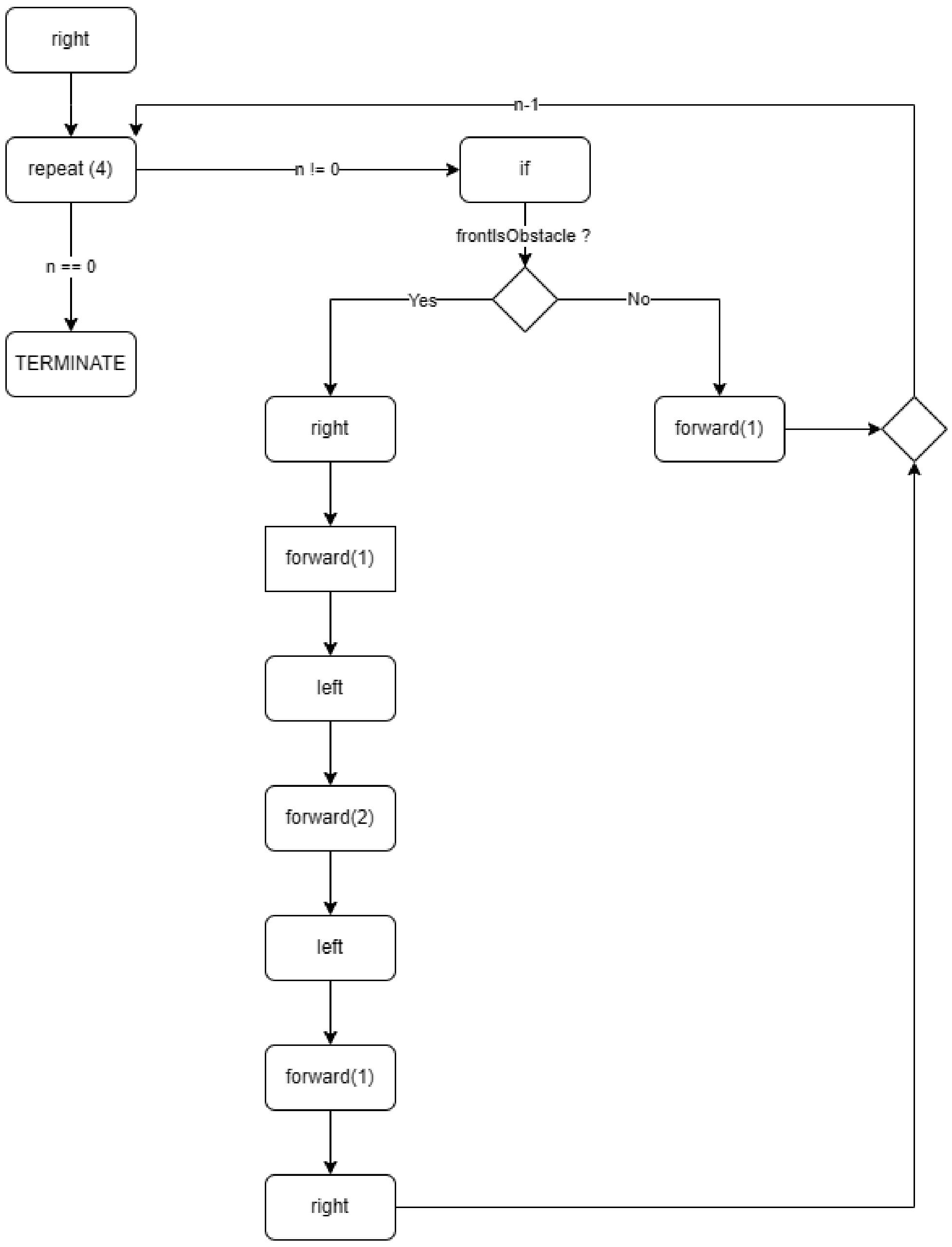
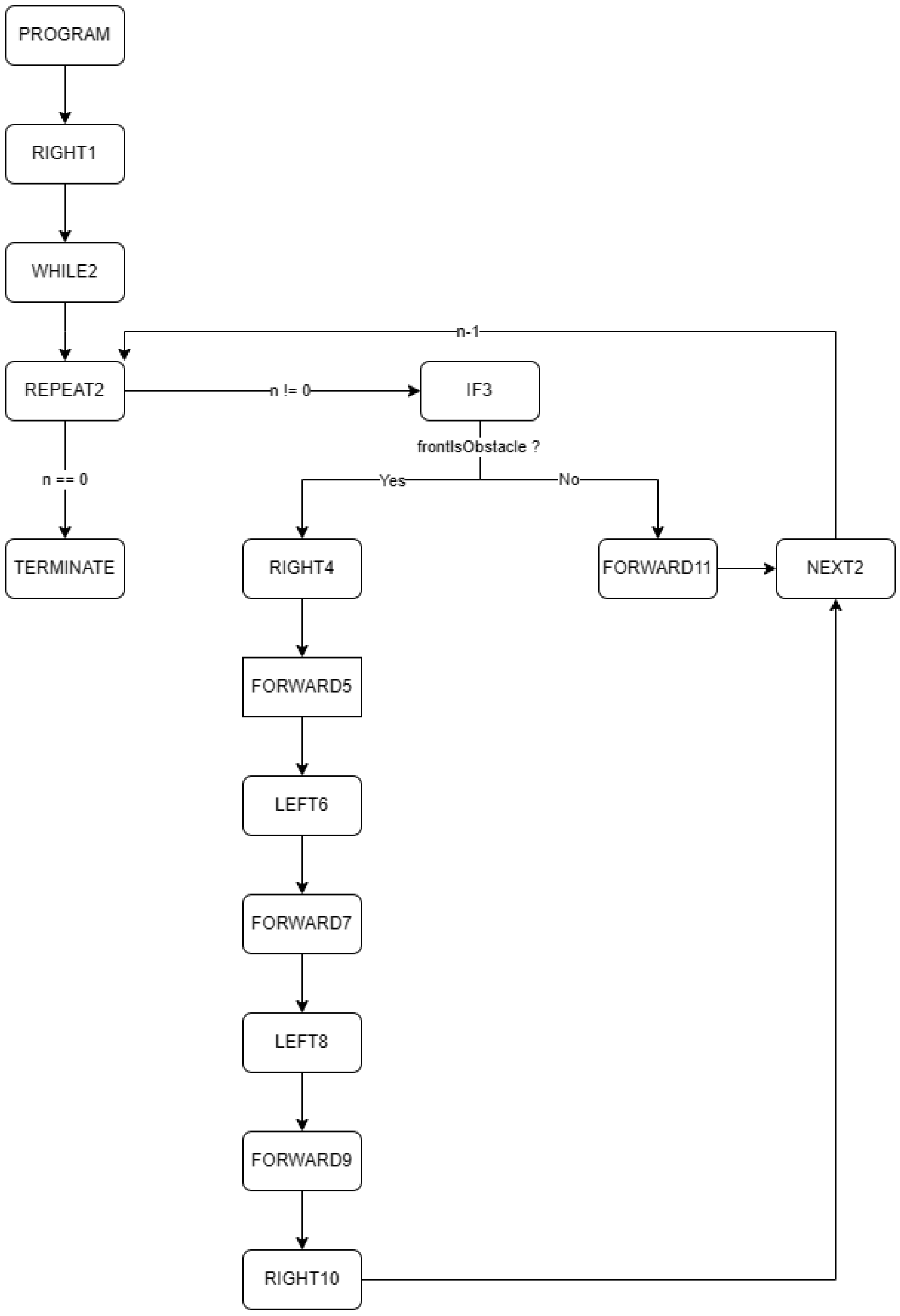


Figure 4 – CSP$_M$ flowchart
Source: Author (2025)



3.2 TRANSLATING A ROBOMIND MAP TO CSP

Our CSP encoding for the map abstracts the visual representation of the walls, representing all possible walls as obstacles. Each of the walls (“C”, “H”, “D”, “G”, ...) is translated to the character “O” that represents an obstacle in the CSP specification. Free spaces (“ ”) are converted to the letter “E”, whereas the robot start position (“@”) translates to the beginning of the configuration (“S”). This systematic encoding makes it possible for the CSP model to mimic the environment. In what follows, we present the CSP model of the map presented in Section 2.1.

```
RAW_MAP = <<O,O,O,O,O,O,O,O,O,O,O,O>,
           <O,O,O,O,O,O,O,O,O,O,O,O>,
           <O,O,S,E,O,E,O,E,O,E,O,O>,
           <O,O,E,E,E,E,E,E,E,E,O,O>,
           <O,O,E,E,E,E,E,E,E,E,O,O>,
           <O,O,O,O,O,O,O,O,O,O,O,O>,
           <O,O,O,O,O,O,O,O,O,O,O,O>>
```

In Figure 5, we show an example of a room that illustrates a bedroom and its dimensions. It has dimensions of 3×3 meters and contains a double bed at the center-north of the room with two bedside tables on either side. At the bottom, we have a wardrobe and a door, which we will consider closed while the robot cleanses the room.

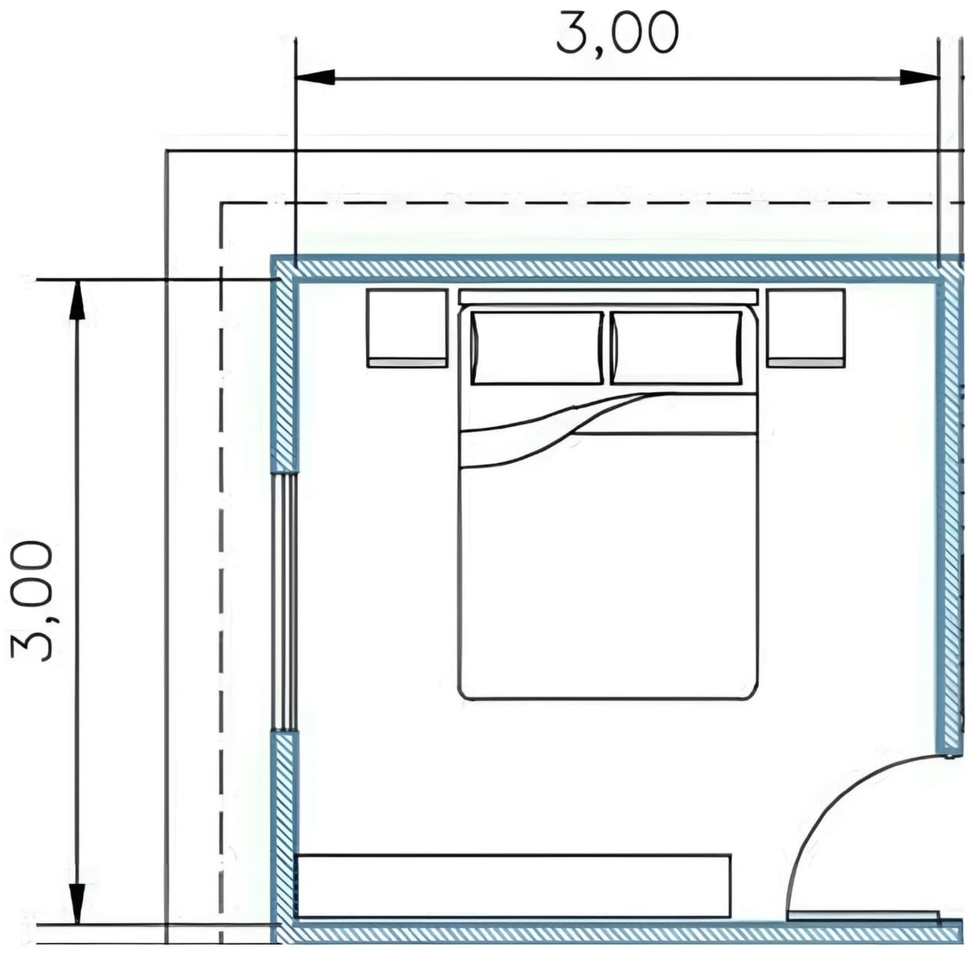
As an AVC has a diameter of about 30 cm, to effectively model the room, we will abstract its dimensions into a 10×10 grid, where each cell represents a square area of 30×30 cm^2, matching the robot size. The bed, bedside tables, and wardrobe will be treated as obstacles, occupying multiple cells depending on their size.

Figure 6 shows the room modeled as a grid. The black cells represent obstacles, and the white cells represent free spaces for the robot. The grid has a dimension of 12×12 cells, which is greater than the original 10×10 grid by the addition of the walls of the room, keeping the central 10×10 grid inside the walls.

Figure 7 shows Figure 5 overlapped by Figure 6, illustrating how the abstraction of the room represents the bedroom. The CSP model for this bedroom is an over approximation. Adjusts in the map size and in the size of the robot (robot radius) can improve the match of the model with the real environment.

We can easily translate the model for the room to CSP. In the translation each black cell is represented as an obstacle (O). We present below the CSP encoding of the map, as presented in Figure 6, using the top-left free space as the starting position.

Figure 5 – Room example
Source: Unknown, Adapted from (UNKNOWN,) (2025)



```
RAW_MAP = <<0,0,0,0,0,0,0,0,0,0,0,0,0>,
           <0,S,0,0,0,0,0,0,0,0,0,E,0>,
           <0,E,0,0,0,0,0,0,0,0,0,E,0>,
           <0,E,E,0,0,0,0,0,0,E,E,E,0>,
           <0,E,E,0,0,0,0,0,0,E,E,E,0>,
           <0,E,E,0,0,0,0,0,0,E,E,E,0>,
           <0,E,E,0,0,0,0,0,0,E,E,E,0>,
           <0,E,E,0,0,0,0,0,0,E,E,E,0>,
           <0,E,E,E,E,E,E,E,E,E,E,E,0>,
           <0,E,E,E,E,E,E,E,E,E,E,E,0>,
           <0,0,0,0,0,0,0,0,0,E,E,E,0>,
           <0,0,0,0,0,0,0,0,0,0,0,0,0>>
```

Figure 6 – Room visualization
Source: Author (2025)

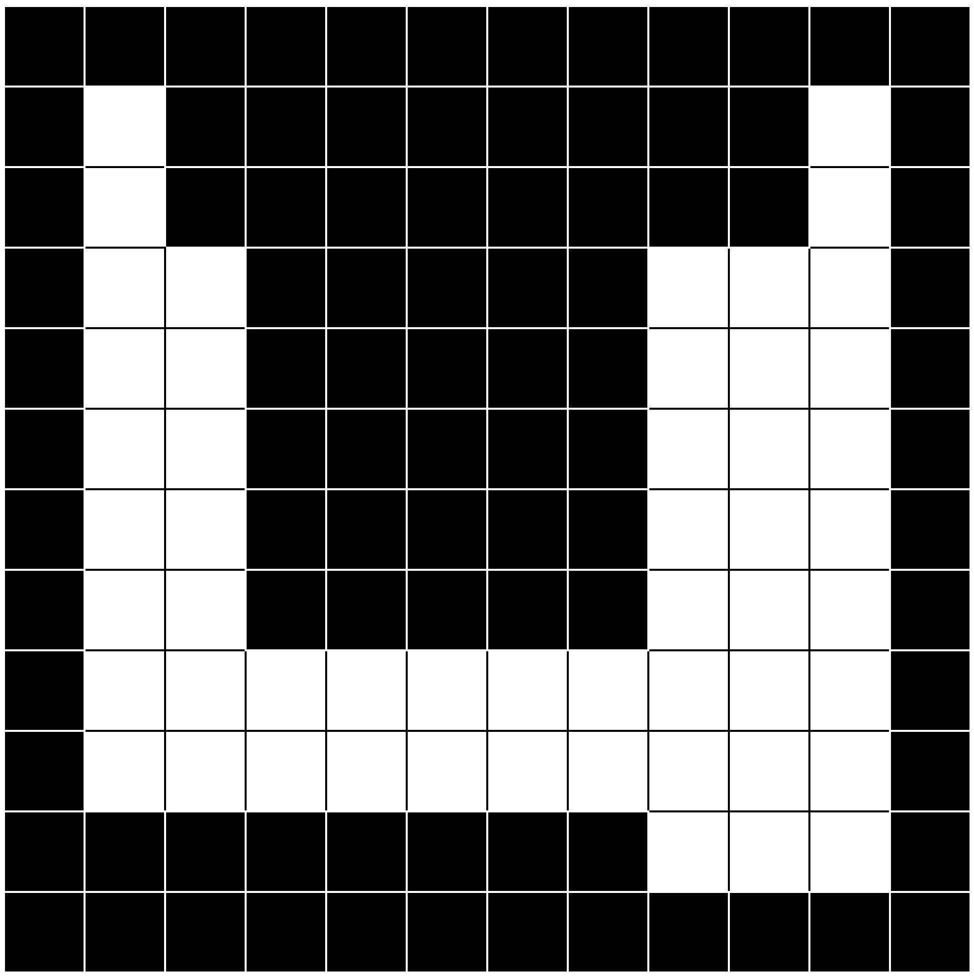
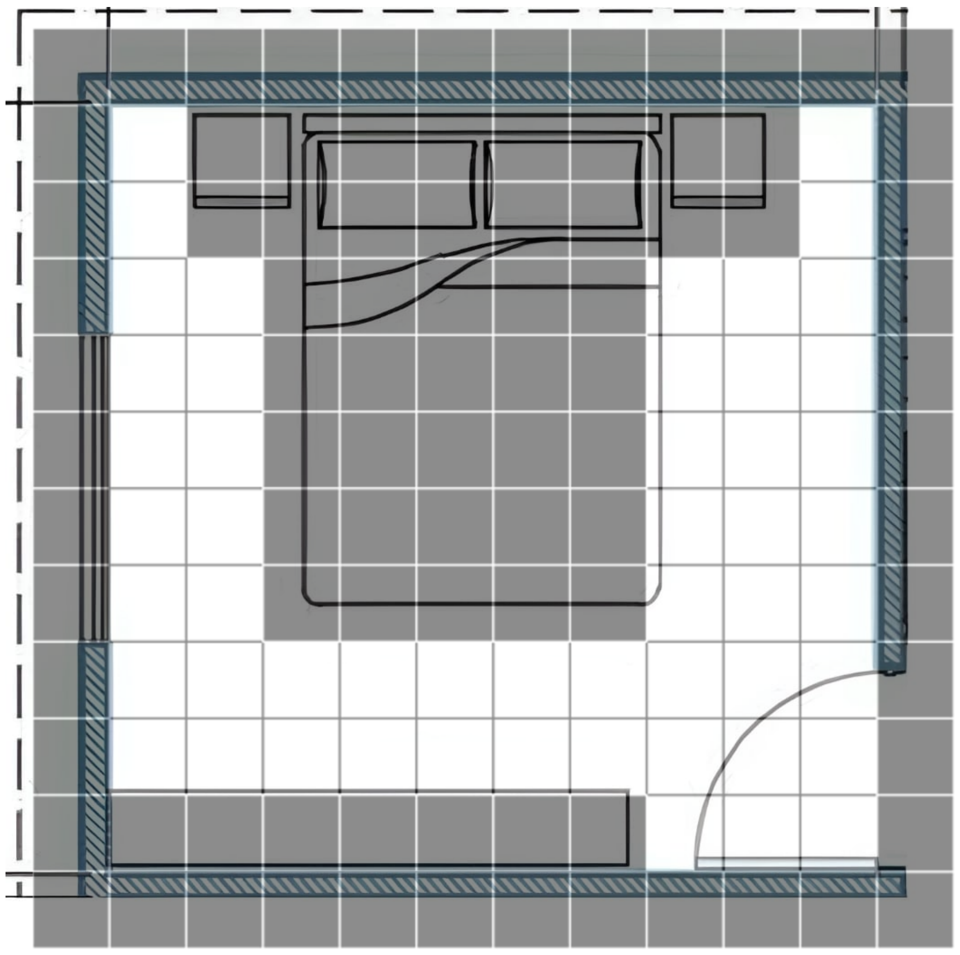


Figure 7 – Overlapping map
Source: Author (2025)



3.3 ROBOMIND ENVIRONMENT MEMORY REPRESENTATION

One important improvement over the formal representation used in (CORREIA, 2021) is the usage of more efficient data structures for representing the program state as well as its environment. The previous work represented a state as a set of tuples that has a linear cost to access and update state values. The current representation has a logarithmic cost instead of a linear one.

The memory of the RoboMind environment is represented in the CSP semantics by a map data structure that stores the names and values of all dynamic elements in the environment. The datatype DATA defines the type for values in the memory as booleans, integers, paint colors, and paint locations.

Another important improvement over the previous work is the inclusion of data structures to assess an AVC. For assessing coverage, we created the constant FREE_MAP to count how many empty spaces exist in the map. We use it to divide the PAINTS set during the refinement to calculate the coverage the robot.

```
datatype Colors = White | Black
datatype DATA = UNDEFINED
              | B.Bool
              | I.Int
              | COLOR.{0..MAXX}.{0..MAXY}.Colors
              | PAINT.Colors

INIT = (|
        X => I.startX,
        Y => I.startY,
        ORIENTATION => I.NORTH_,
        IS_PAITING => B.true,
        PAINT_COLOR => PAINT.White,
        EFFORT => I.0
       |)

PAINTS = { ((startX,startY), White) }
```

3.4 FORMAL DEFINITIONS OF COVERAGE AND EFFORT

To quantitatively evaluate the robot's cleaning performance, we define three key metrics used in this work to evaluate cleaning algorithms:

- `Effort` is defined as the number of steps (squares moved) performed by the robot;
- `Turns` is defined as the amount of $90ž$ turns performed by the robot; and

- Coverage is calculated by the formula N_{clean_cells} / N_{free_cells}, such that N_{clean_cells} denotes the total number of cells cleaned and N_{free_cells} denotes the total number of free cells.

Other important efficiency factors are not considered in this approach, such as time taken by the robot and energy consumption through the path-planning. However, as energy consumption and time taken are directly proportional to the robot's movement, making steps and turns reasonable proxies for efficiency.

Another important efficiency factor not included in this approach is the amount of dirtiness in a grid cell, which captures the user experience and satisfaction with the robot's performance. While this strategy ensures a standardized evaluation for coverage path-planning algorithms, potential future enhancements can include evaluating how well the robot meets perceived cleanliness standards beyond raw coverage calculations.

In our CSP$_M$ specification, we introduced a dedicated channel named coverage, which is triggered each time the robot cleans a new grid cell. The event follows the format `coverage.x`, where $(0 \leq x \leq 100)$ represents the percentage of coverage at that point in the execution. This allows us to monitor progress via event traces. During model-checking, we apply the following refinement assertion:

```
assert STOP [T= PROGRAM \diff(Events, {|coverage|})
```

This assertion ensures that if any `coverage.x` event is emitted, it becomes a counterexample trace, enabling us to track how much of the map has been cleaned before termination. By analyzing the last `coverage.x` in the trace, we can determine the maximum coverage achieved by the algorithm.

3.5 ROBOT SIZE AND PATH FEASIBILITY

To describe the grid map and the robot size more accurately, we introduced the constant R (radius) to describe the robot's size, which does not vary during the test. This constant helps to set the diameter of the robot and check inaccessible paths, which are narrower than the movement step of the robot.

Figure 7 illustrates what we will represent as a robot with a zero radius, where it sets the robot dimension equal to the size of the grid cell. With a 30 cm $\times$ 30 cm cell, some cells are left for us to interpret whether it is an obstacle.

If we reduce the size of each cell, we can reduce the map's obstacle abstraction. With a radius value of zero, the robot fits in a 1×1 cell. If we increase the radius to one, we want the robot to fit more cells, but we do not want to change the validation. In order to reduce the amount of changes for the validation, we will fit the robot center in a cell instead of a corner of a cell. So, we cannot divide each cell by two along both x-y dimensions and have four cells for each cell, as the center of the robot will be in a corner. If the robot center occupies a cell, a robot with a radius of 1 occupies a total of 3×3 cells. By dividing each cell by three along both x-y dimensions, each new cell will have a size of 10 cm $\times$ 10 cm instead of 30 cm $\times$ 30 cm. With a cell size of 10 cm $\times$ 10 cm, we will have the robot fitting the center of the cell and still occupying the 30 cm $\times$ 30 cm.

Figure 8 illustrates the visual representation of the Figure 7 for a robot with a radius value equal to one, where the robot will fit in a 3×3 grid area of the map. In contrast to Figure 7, most of the cells we judged as obstacles are now free spaces, as the cell size allows us to judge better if the cell is or not an obstacle.

Some minor changes had to happen to assess the robot's new size. Below, we present an example of how obstacle detection works without increasing the robot's size.

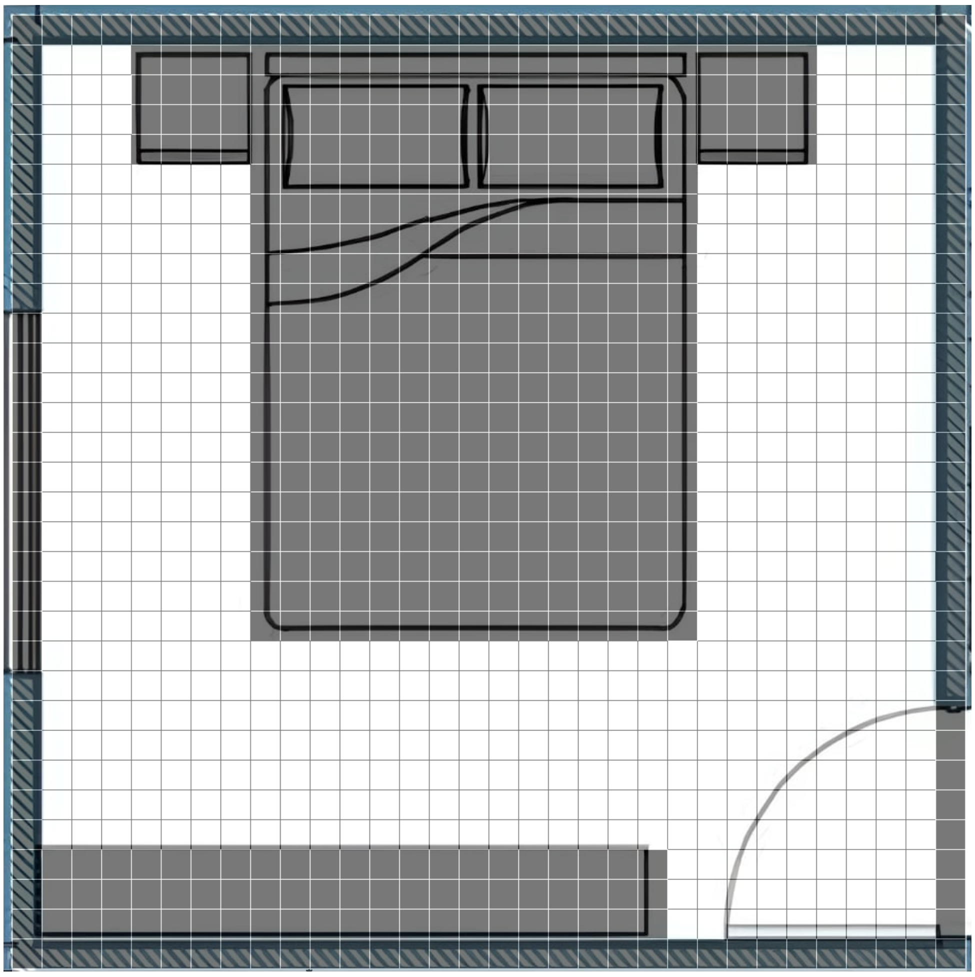
```
frontIsObstacle(x,y,o) = thingsInFront(x,y,o) == {0}
frontIsClear(x,y,o) = not(frontIsObstacle(x,y,o))
```

To check if the robot's path is clear, we call for the `frontIsClear()` function, which calls for the `frontIsObstacle()`, checking in the map if there is an obstacle. If an obstacle is found, the `thingsInFront` function will return `{0}`, which will validate the presence of an obstacle ahead of the robot. If there is no obstacle ahead of the robot, the function will return `{}`. For the `leftIsClear` and `rightIsClear` functions, we call for the `frontIsClear` function, adjusting the orientation for the direction.

To check now if the path is clear for the new robot size, we have to rely on the `frontIsClear`, but for the new size. For this, we created a new function called `allFrontIsClear`, which calls for the `frontIsClear`, adjusting the front with the radius. Below, we present how the `allFrontIsClear` works.

```
allFrontIsClear(r, x, y, o) =
  if(o == NORTH_) then (
      frontIsClear(x-r, y-RADIUS, o)
```

Figure 8 – Map with a grid-cell size of 10×10 cm
Source: Author (2025)



```
      and frontIsClear(x+r, y-RADIUS, o)
      and allFrontIsClear(r-1, x, y, o)
  ) else if (o == EAST_) then (
      frontIsClear(x+RADIUS, y-r, o)
      and frontIsClear(x+RADIUS, y+r, o)
      and allFrontIsClear(r-1, x, y, o)
  ) else if (o == SOUTH_) then (
      frontIsClear(x-r, y+RADIUS, o)
      and frontIsClear(x+r, y+RADIUS, o)
      and allFrontIsClear(r-1, x, y, o)
  ) else (
      frontIsClear(x-RADIUS, y-r, o)
      and frontIsClear(x-RADIUS, y+r, o)
      and allFrontIsClear(r-1, x, y, o)
```

The `allFrontIsClear` function relies on the `RADIUS` constant, which is the actual radius of the robot, and checks with the `frontIsClear` function if, for each of the cells occupied in front of the robot, if there is an obstacle in all the cells ahead. If, for the actual value of the radius, there is not an obstacle, the function recursively calls itself with the radius received reduced.

To assess the painting, which we use to validate coverage, we use the PAINTWHITE / PAINTBLACK process. The PAINTWHITE is shown below.

```
PAINTWHITE(m, paints, next) =
  let
    x = get(m,X)
    y = get(m,Y)
    m_ = mapUpdateMultiple(m, <(IS_PAITING, B.true),(PAINT_COLOR,
   PAINTS.White)>)
  within
    next(m_, union(paints, {((x, y), White)}))
```

The PAINTWHITE process paints only the robot's actual position. Some minor changes were made to make it paint with the radius. First, we needed to modify the painting function so that it could both paint for the basic robot (where it occupies one cell) and the new robot size (where it occupies nine cells). To find the nine cells occupied by the root, we introduced a new function, PAINTBYRADIUS.

```
PAINTBYRADIUS(x, y, color) =
  let
    PAINTBYX(y, x, xlim) = if(x == xlim) then <((x, y), color)> else
   <((x, y), color)>^PAINTBYX(y, x+1, xlim)
    PAINTBYY(y, ylim) = if y == ylim then PAINTBYX(y, x-RADIUS, x+
   RADIUS) else PAINTBYX(y, x-RADIUS, x+RADIUS)^PAINTBYY(y+1, ylim)
  within
    set(PAINTBYY(y-RADIUS, y+RADIUS))
```

The PAINTBYRADIUS function goes line by line, collecting each cell that needs painting and returning the collection/set of these cells. The function calls for the PAINTBYY function, which receives two arguments: the actual y and the limit ylim, which is the maximum value y can have before ending the recursion. Each time PAINTBYY is called, it calls for the PAINTBYX function. The PAINTBYX function receives y, x, and xlim, much like PAINTBYY function. for each x and y, it registers the color to paint in a sequence and starts a recursion with x+1 until x is equal to xlim.

With the PAINTBYRADIUS function, we can edit the PAINTWHITE function to call for the PAINTBYRADIUS

```
PAINTWHITE(m, paints, next) =
  let
    x = get(m,X)
    y = get(m,Y)
    m_ = mapUpdateMultiple(m, <(IS_PAITING, B.true),(PAINT_COLOR,
   PAINTS.White)>)
  within
    next(m_, union(paints, PAINTBYRADIUS(x, y, White))
```

In Chapter 4, we will experiment with how the assessment works with a robot of radius value one after experimenting with how the radius zero behaves.

3.6 DETERMINING THE MAXIMUM POSSIBLE COVERAGE

In order to determine coverage using the FDR refinement checker within a defined time limit, we apply two distinct but interrelated verification methods. Both use the following traces refinement verification condition, where PROGRAM represents the ROBO code translated into CSP$_M$ following the approach introduced in (CORREIA, 2021). Such a refinement verifies if PROGRAM (with all events removed except for covered) behaves as STOP. As STOP does not produce any event at all, this refinement fails if the process PROGRAM \diff(Events, |covered|) produces events covered.x, such that x $(0 \leq x \leq 100)$ records the coverage percentage.

```
assert STOP [T= PROGRAM \diff(Events, {|covered|})
```

The first verification method uses the previous trace refinement once intending to achieve 100% coverage within a 1 hour timeout $(3600$ seconds). If FDR provides counterexamples, the second verification is unnecessary. If FDR fails to provide a counterexample, we proceed to the second, more intensive verification method, which leverages a search algorithm (Algorithm 1), using the previous traces refinement several times if necessary.

3.6.1 Single verification method

FDR applies the breadth-first search strategy in a state-machine graph to obtain the shortest counterexample, which might not present the maximum coverage as the shortest counterexample. To ensure we can observe the highest possible coverage, we set FDR to produce all counterexamples that make it possible to look for the maximum value for coverage record in the counterexample traces.

3.6.2 Binary search-based verification method

Given that FDR fails to produce all counterexamples within the time constraint using the refinement presented in the previous section, we proceed with a more computationally intensive verification method based on a binary search. This process follows the steps outlined in Algorithm 1.

Algorithm 1 CBinarySearch

```
Require: minimum, maximum
Ensure: result
 1: if $maximum - minimum \leq 1$ then
 2:     $result \leftarrow minimum$
 3: end if
 4: $actual \leftarrow (minimum + maximum)/2$
 5: $save(\text{“}script.csp\text{”}, GenerateScript(actual))$
 6: $coverages \leftarrow run([\text{“}fdr\text{”}, \text{“}script.csp\text{”}], 3600)$
 7: if $isNotEmpty(coverages)$ then
 8:     if $max(coverages) \geq actual$ then
 9:         $new \leftarrow max(coverage)$
10:         $result \leftarrow CBinarySearch(new, maximum)$
11:     else
12:         $result \leftarrow max(coverages)$
13:     end if
14: else
15:     $result \leftarrow CBinarySearch(minimum, actual)$
16: end if
```

In Algorithm 1, we generate a CSP$_M$ specification in `GenerateScript` that will end when it attains the given coverage. For example, in the first iteration of `CBinarySearch`, where `actual` is 50%, the refinement will end when `PROGRAM` reaches 50% coverage. In `isNotEmpty`, we check if the array `coverages` is not an empty array. If it is an empty array (when it reaches the time constraint), it returns `false`. If it has any value, it returns `true`. In `max`, we return the maximum value from the array that records the `coverages`. We explain each line of the algorithm in what follows:

- In Lines 1 and 2, if the difference between `maximum` and `minimum` is less than or equal to 1, then the binary search has reached its final iteration. At this point, the function returns the `minimum` value;
- In Line 4, it computes the mean value between `minimum` and `maximum` for the search as `actual` (`C`);
- In Line 5, the algorithm generates a file containing the refinement to verify if the code can reach `C`;

- In Line 6, the algorithm executes FDR, and the counterexamples (`Ce`) yield by the refinement process are collected;
- In Line 7, the script checks if any counterexamples (`Ce`) were generated:
  - If any `Ce` were generated, it checks if they meet or exceed `C`. Given that the maximum `Ce` is unknown, it can be either greater than, equal to, or less than `C`;
    - If the highest value of `Ce` exceeds `C`, it indicates that the map is relatively small, and each robot step increases coverage by more than 1%;
    - If the highest value of `Ce` is equal to `C`, this scenario mirrors the standard behavior of a binary search;
    - If the highest value of `Ce` is below `C`, this suggests that the highest possible coverage for the algorithm is the highest value of `Ce`.
  - If any generated `Ce` meets `C`, recursion is initiated in Line 10 using a new `minimum` value set to the value `max(coverages)` - indicating that this level reached the desired coverage within the timeout constraint;
  - If the generated `Ce` fails to meet `C`, `C` is impossible to be attained by the current algorithm. In this case, the function returns the maximum value within `Ce` in Line 12;
- If no `Ce` were generated, the maximum coverage is adjusted to `actual` in Line 15 since no `Ce` were found.

4 EVALUATION

We evaluated four different CPP algorithms (**Code**$_1$, **Code**$_2$, **Depth-First Search**, and **Inwards Spiral**) across various map configurations. The deterministic algorithms (**Depth-First Search** and **Inwards Spiral**) are compared against non-deterministic algorithms (**Code**$_1$ and **Code**$_2$) to assess their performance trade-offs.

The evaluations were validated in Section 4.6.1 and 4.6.2 and conducted on an AMD Ryzen 7 3700X system with 24 GB RAM, Windows 11 Pro x64, and a 1TB NVMe SSD. Results compare analysis time, coverage, and robot turns and effort, highlighting the strengths and limitations of each strategy.

4.1 MAP SELECTION PROCESS

The map selection of this work involved choosing maps that compose all the rooms in a household environment, ensuring a comprehensive representation of domestic environments for the AVC's operation. The maps were collected from a series of house plants and translated to grid-based maps. The maps selected were a living room, a kitchen, and three bedrooms. Multiple bedrooms may seem redundant when talking about cleaning efficiency. However, real-world homes include multiple rooms, so evaluating how the vacuum adapts to different room scenarios is an important evaluation to be explored in future work.

4.2 CODE$_1$ IN ROBO

The following code approximates an AVC algorithm, using white paint to mark covered locations. Inspired by observing the Roomba's (CORPORATION, 2025) random bounce behavior, the algorithm below assumes sensor-based path recording. Its implementation shows to behave as:

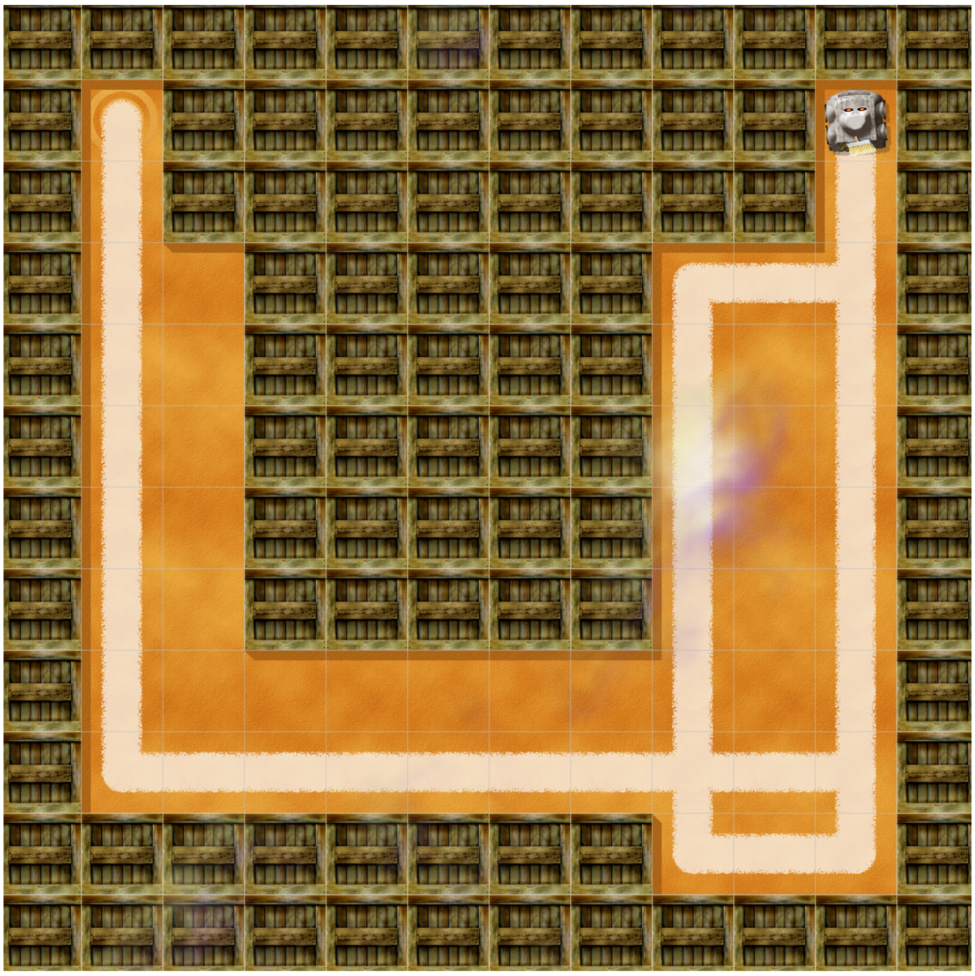
1. Go forward
2. Hit an obstacle
3. Turn to a different direction
4. Return to step 1

```
paintWhite()
repeat{
  repeatWhile(frontIsClear()){
    forward(1)
  }
  if(flipCoin()){
    if(flipCoin()){
      north(0)
    } else {
      south(0)
    }
  } else {
    if(flipCoin()){
      east(0)
    } else {
      west(0)
    }
  }
}
```

By adopting this behavior, **Code**$_1$ seeks to replicate the random bounce behavior, utilizing sensor-based path planning to effectively navigate and clean an area.

Figure 9 illustrates the routes of the robot that executes the **Code**$_1$ algorithm. Due to the algorithm's simplicity, gaps are left unreachable by the robot, resulting in a coverage rate below 100%. The untouched areas should be exacerbated on larger maps, reducing coverage efficiency.

Figure 9 – Code$_1$ visualization
Source: Author (2025)



4.3 CODE$_2$ IN ROBO

The following code snippets comprise auxiliary functions within the primary **Code**$_2$ algorithm. **Code**$_2$ is an improvement of the **Code**$_1$ random bounce's algorithm, where we avoid bouncing to already clean cells.

Unlike **Code**$_1$, **Code**$_2$ halts when no suitable paths remain. This algorithm performs a look-ahead, continuing the navigation until it attains a predefined number of steps, improving the cleansing of the room.

```
cango = true

paintWhite()

repeatWhile(canGo){
  nic = northIsClear()
  sic = southIsClear()
  eic = eastIsClear()
  wic = westIsClear()
  canGoVert = nic or sic
  canGoHor = eic or wic
  if(not(canGoVert or canGoVor)){
    canGo = newWay()
  }else{
    chooseWay()
    repeatWhile(frontIsClear()){
      forward(1)
    }
  }
}
```

The code below presents a ROBO auxiliary procedure used in the definition of **Code**$_2$. There are auxiliary procedures for each direction in which the robot can move. Each procedure involves rotating the robot in a specific direction and then checking for potential obstacles ahead or any markings on the ground that indicate already cleaned areas. If the path is blocked, it is unsuitable for further exploration, returning False. If it is free, it returns True. For instance, the procedure northIsClear turns the robot to the north direction and then checks if in the north direction there is no obstacle and it is not painted over. If both are true, then it returns True. Otherwise, it returns False.

```
procedure northIsClear{
  north(0)
  return frontIsClear() and not(frontIsWhite())
}
```

The code presented below defines the chooseWay function. This function evaluates whether a vertical or horizontal direction is obstructed before invoking the flipCoin function. Its design aims to optimize the decision-making process by limiting the invocation of flipCoin to instances where it is strictly necessary. Once a direction is defined, the function delegates the decision to the choose function for further processing.

```
procedure chooseWay(){
  if(cangovert and not(cangohor)){
    vert = true
  } else if(cangohor and not(cangovert)){
    vert = false
  } else {
    vert = flipCoin()
  }
  if(vert){
    if(northIsClear() and not(southIsClear())){
      north(0)
    }else {
      if (southIsClear() and not(northIsClear())){
        south(0)
      } else {
        if(flipCoin()){
          north(0)
        } else {
          south(0)
        }
      }
    }
  } else {
    if(eastIsClear() and not(westIsClear())){
        east(0)
    }else {
      if (westIsClear() and not(eastIsClear())){
        west(0)
      } else {
        if(flipCoin()){
          east(0)
        } else {
          west(0)
        }
      }
    }
  }
}
```

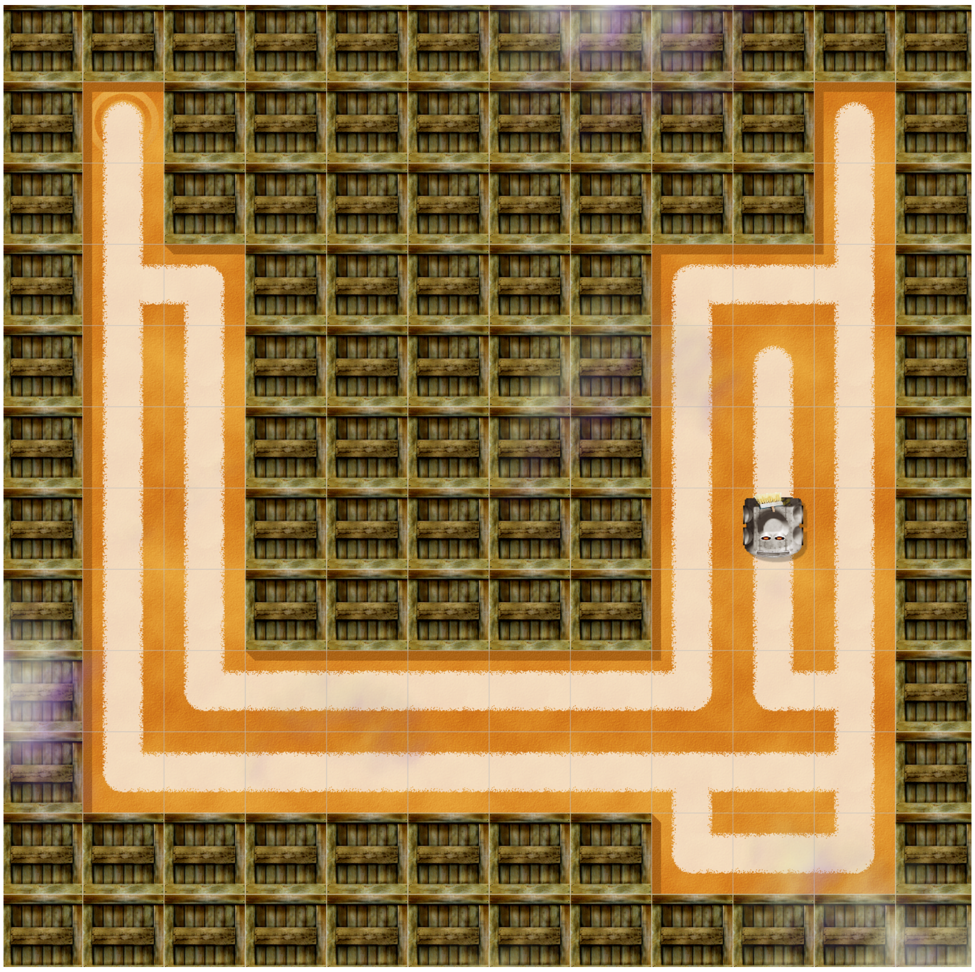
The code below introduces the newWay function, which exhibits greater complexity when compared to the previously discussed functions. Its primary purpose is to identify unpainted sections on the map to paint. The function iteratively searches for an unpainted cell until it locates a suitable cell or the counter is greater or equal to the maximum steps threshold.

During each iteration, the function selects a direction and moves forward, scanning the front, left, and right for unpainted areas. Upon finding an unpainted section, it changes its direction accordingly, terminates the procedure, and returns to **Code**$_2$, standing in front of the unpainted area and signaling the discovery of an unpainted section. Conversely, if no unpainted cell was found within the step limit, the function concludes that the unpainted section is inaccessible from the current position.

```
procedure newWay(){
  count = 0
  found = false
  repeatWhile(count < steps and not(found)){
    nic = northIsClear()
    sic = southIsClear()
    eic = eastIsClear()
    wic = westIsClear()
    cangovert = nic or sic
    cangohor = eic or wic
    chooseWay()
    repeatWhile(frontIsClear() and count < steps){
      forward(1)
      count = count + 1
      if(frontIsClear() and not(frontIsWhite())){
        found = true
        break
      }else if(rightIsClear() and not(rightIsWhite())){
        right()
        found = true
        break
      }else if(leftIsClear() and not(leftIsWhite())){
        left()
        found = true
        break
      }
    }
  }
  return(found)
}
```

Figure 10 illustrates the behavior of **Code**$_2$ algorithm. Unlike **Code**$_1$, **Code**$_2$ demonstrates a more comprehensive cleaning pattern, continually exploring new areas to clean. This proactive approach results in significantly higher coverage compared to its predecessor.

Figure 10 – Code$_2$ visualization
Source: Author (2025)



4.4 DEPTH-FIRST SEARCH

The Depth-First Search (DFS) algorithm performs a systematic exploration delving as deeply as possible into a given search space before backtracking. When applied to AVCs, DFS ensures that the robot methodically explores an area, thoroughly covering one path before retreating and selecting an alternative direction to follow. The algorithm adopts a stack-based traversal approach, moving in one direction until encountering an obstacle or a previously visited cell. When this happens, it backtracks to the last available branch and resumes exploration. This process leads to a structured but non-optimal coverage path, as Depth-First Search (DFS) does not inherently prioritize efficiency or minimal turns. However, it ensures a complete traversal of the available cleaning area.

```
paintWhite()
repeat(4){
  if(frontIsClear()){
    break
  }
  right()
}
dfs()
```

The DFS algorithm below follows a recursive approach, systematically exploring the cleaning space. It starts by marking its initial position (`paintWhite()`) and rotates to a clear direction before calling the recursive function `dfs()`.

The function `dfs()` is structured as follows:

- If the front is clear and uncleaned, the robot moves forward, marks the cell, and recursively calls `dfs()` to maintain linear motion before backtracking.
- Then, if the left side is clear and uncleaned, the robot turns left, moves forward, and applies `dfs()` before backtracking and restoring orientation.
- Lastly, if the right side is clear and uncleaned, the robot turns right, moves forward, and applies `dfs()` before backtracking and restoring orientation.

This algorithm ensures full exploration of the environment, backtracking when necessary to cover missed areas. However, the backtracking mechanism may increase effort and analysis time, making DFS less efficient in environments with multiple obstacles.

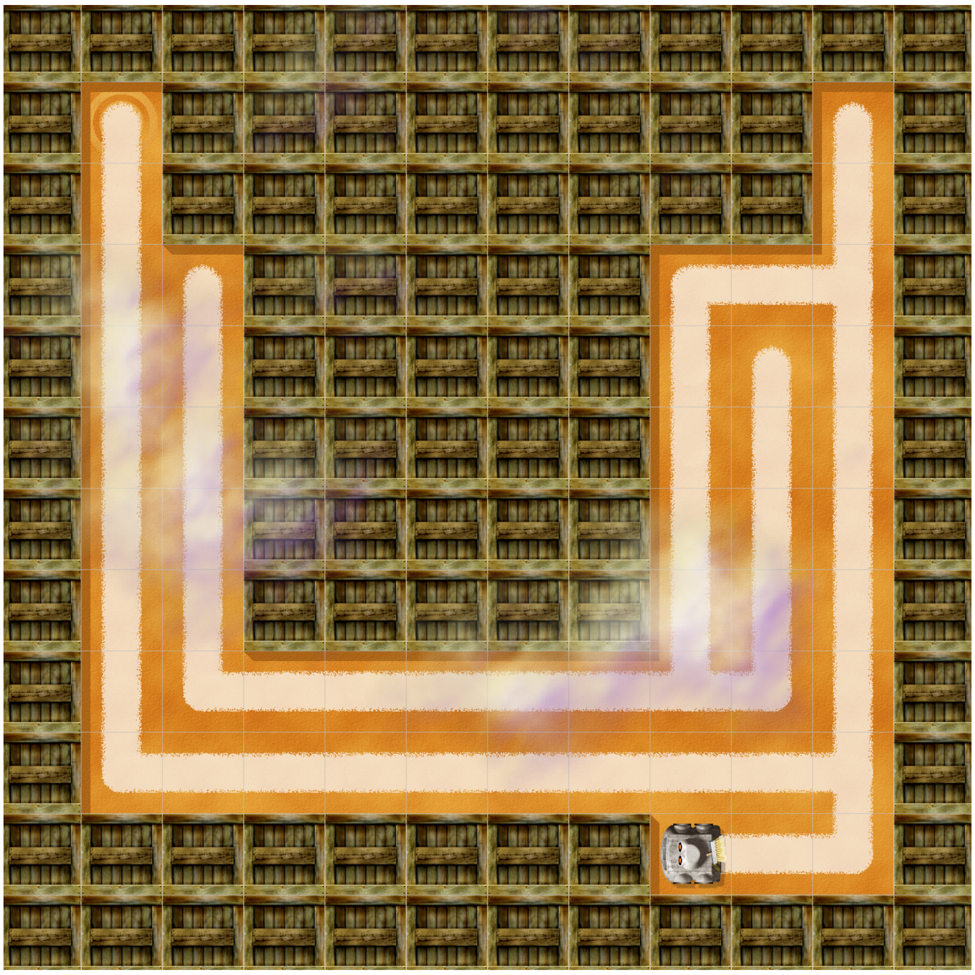
```
procedure dfs(){
  if(frontIsClear() and not(frontIsWhite())){
    forward(1)
    dfs()
    backward(1)
  }

  if(leftIsClear() and not(leftIsWhite())){
    left()
    forward(1)
    dfs()
    backward(1)
    right()
  }

  if(rightIsClear() and not(rightIsWhite())){
    right()
    forward(1)
    dfs()
    backward(1)
    left()
  }
}
```

Figure 11 illustrates the behavior of **Depth-First Search** algorithm. Unlike **Code**$_1$ and **Code**$_2$, it demonstrates a deterministic cleaning pattern.

Figure 11 – DFS visualization
Source: Author (2025)



4.5 INWARDS SPIRAL

The Inwards Spiral algorithm is a structured CPP approach that prioritizes efficiency by minimizing redundant movements and unnecessary turns. Instead of randomly navigating the space, the robot follows a predetermined spiral trajectory, starting from the outer edges of the environment and gradually moving inward. This method reduces the number of abrupt turns and ensures that the cleaning path remains compact, covering the largest continuous area before adjusting direction. Inwards Spiral is particularly effective for open environments with fewer obstacles, as it allows for smooth and efficient coverage without excessive maneuvering. However, with rooms with complex layouts or numerous obstacles, this approach may require adaptations to handle navigation constraints effectively.

```
paintWhite()
repeat(4){
  if(frontIsClear()){
    break
  }
  right()
}
spiral()
```

The Inwards Spiral algorithm below follows an efficient approach to cover the cleaning area by prioritizing rightward turns, ensuring a structured and systematic cleaning approach. Unlike **DFS**, which explores the deepest path first, this algorithm follows a controlled turning strategy to Inwards Spiral, reducing unnecessary backtracking.

The algorithm begins by marking the initial position as cleaned (`paintWhite()`) and rotates to a clear direction before calling the recursive function `spiral()`.

The function `spiral()` is structured as follows:

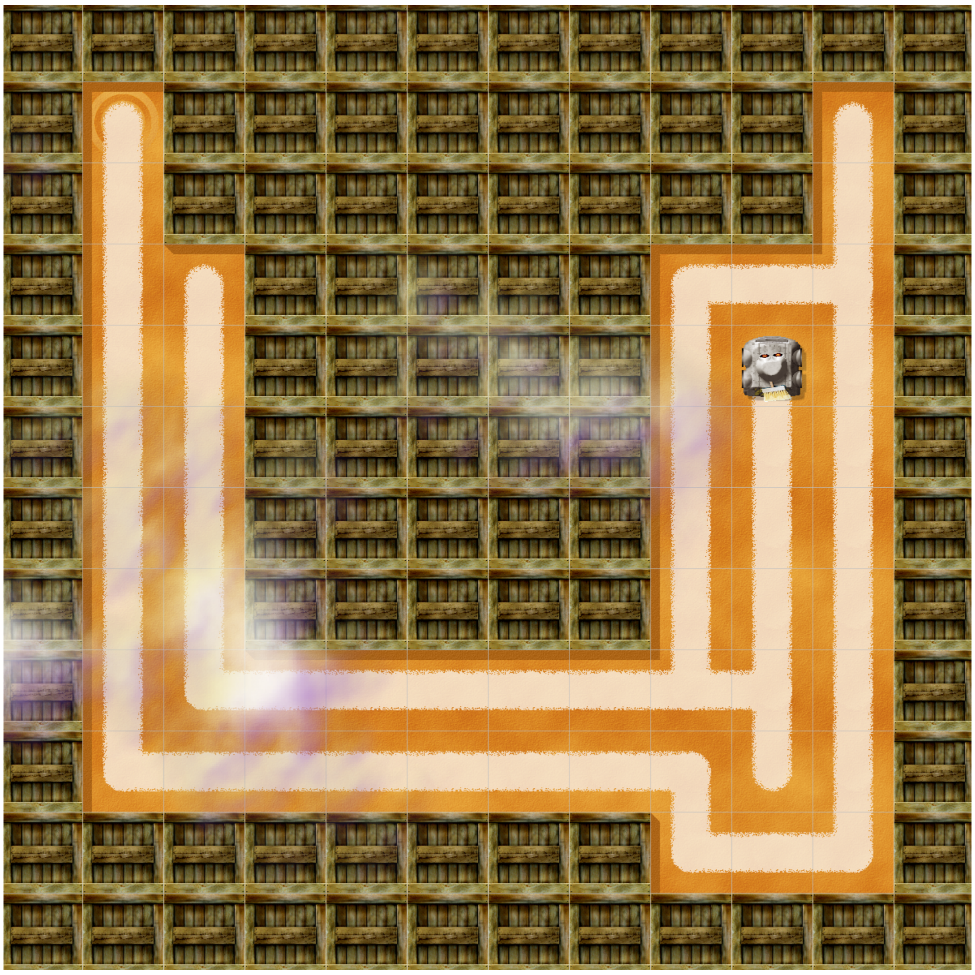
- If the right side is clear and uncleaned, the robot turns right, moves forward, and recursively calls `spiral()` to continue in the new direction. After backtracking, it turns left to restore orientation.
- If the front is clear and uncleaned, the robot moves forward and recursively calls `spiral()` to maintain linear motion before backtracking.
- If the left side is clear and uncleaned, the robot turns left, moves forward, and recursively calls `spiral()` before backtracking and restoring orientation.

This approach naturally guides the robot in a spiraling motion, efficiently covering the map by prioritizing turns and ensuring all reachable areas are cleaned. However, in environments with irregular obstacle placement, the spiral pattern will interrupt, requiring additional adaptations to navigate around obstacles effectively.

```
procedure spiral() {
  if(rightIsClear() and not(rightIsWhite())){
    right()
    forward(1)
    spiral()
    backward(1)
    left()
  }
  if(frontIsClear() and not(frontIsWhite())){
    forward(1)
    spiral()
    backward(1)
  }
  if(leftIsClear() and not(leftIsWhite())){
    left()
    forward(1)
    spiral()
    backward(1)
    right()
  }
}
```

Figure 12 illustrates the behavior of **Inwards Spiral** algorithm. It demonstrates a linear cleaning pattern, exploring new areas to clean when it returns to its original position, just like DFS.

Figure 12 – Spiral visualization
Source: Author (2025)



4.6 RESULTS AND COMPARISON

In this section, we present and analyze the performance results from our approach comparing **Code**$_1$, **Code**$_2$, **Depth-First Search** and **Inwards Spiral** across various map configurations. Our analysis emphasizes FDR verification time, effort, turns, and coverage, with the objective being to evaluate the effectiveness of each strategy in different environments using the strategy introduced in the previous section.

4.6.1 Approach Validation

In this section, we will analyze the performance results considering the behavior of the single verification method with a single robot occupying 1×1 cells of the grid. Tables 4 and 5 in this section are organized as follows: the first column specifies the **Map** dimensions with Rows (R) and Columns (C) represented as R $\times$ C. The following column is divided into four sections, one for each algorithm. Each of the following sections is organized as follows: the first column shows the average verification time in seconds, calculated from 30 independent runs, alongside the standard deviation as a measure of variability (average on the left, standard deviation on the right). The second and third columns exhibit the robot's effort and turns, indicating the computational or operational load involved. The fourth column (**%**) displays the achieved coverage within a 1-hour limit for the verification that runs refinement assertions using FDR. If the refinement fails to produce a result within this time limit, the verification terminates, and a minus ($-$) symbol indicates that the refinement attained the time constraint limit.

4.6.1.1 Optimal Global Coverage

Table 4 presents the performance metrics of all four algorithms across five maps, which collectively model a simulated household environment.

Table 4 – Optimal global coverage overview for single robots

Map	**Code$_1$**				**Code$_2$**			
	Time (s)	Effort	Turns	%	Time (s)	Effort	Turns	%
Living Room (19×12)	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
Kitchen (12×8)	39.3 ± 1.3	50	47	96	1.1 ± 0.1	31	60	100
Room 1 (12×12)	15.7 ± 0.7	60	42	69	2.0 ± 0.1	31	85	100
Room 2 (14×11)	$-$	$-$	$-$	$-$	64.2 ± 0.8	52	81	100
Room 3 (12×11)	996.3 ± 20.3	49	72	88	10.9 ± 0.2	42	62	100
Map	**Depth-First Search**				**Inwards Spiral**			
	Time (s)	Effort	Turns	%	Time (s)	Effort	Turns	%
Living Room (19×12)	0.4 ± 0.1	139	51	100	0.5 ± 0.1	147	60	100
Kitchen (12×8)	0.3 ± 0.1	51	14	100	0.2 ± 0.1	30	10	100
Room 1 (12×12)	0.3 ± 0.1	83	18	100	0.3 ± 0.2	66	15	100
Room 2 (14×11)	0.3 ± 0.1	70	21	100	0.3 ± 0.1	64	15	100
Room 3 (12×11)	0.3 ± 0.1	72	22	100	0.2 ± 0.1	42	11	100

Comparing the overall results of the algorithms, **Code$_1$** failed to finish the validation in the Living Room and Room 2 within 1 hour, while **Code$_2$** only failed to reach full coverage in the Living Room. All other maps and algorithms concluded successfully, reaching up to 100% coverage.

When comparing all the algorithms in Table 4, we have the following points:

- **Verification Time**: Both **DFS** and **Inwards Spiral** were the most efficient to analyse, completing the task in less than one second for all maps. **Code**$_2$ failed to find a solution within the one-hour time limit for the Living Room, but achieved times between one second and one minute for the other maps. **Code**$_1$ failed to obtain results for the Living Room and Room 2, being the more costly algorithm overall, with times ranging from 15 seconds to approximately 16 minutes.
- **Coverage**: The **DFS** and **Inwards Spiral** algorithms achieved 100% coverage on all maps. **Code**$_2$ also reached 100% in every map except for the Living Room. On the other hand, **Code**$_1$ did not achieve 100% coverage on any of the maps, with values ranging from 69% (Room 1) to 96% (Kitchen).
- **Effort**: **Code**$_2$ had the lowest total effort on most maps, losing by only one point to Inwards Spiral in the Kitchen. The overall ranking for the lowest effort was: **Code**$_2$, followed by **Inwards Spiral**, **Depth-First Search**, and lastly, **Code**$_1$. Since **Code**$_1$ did not reach 100% in any map, it cannot be directly compared to the others. For example, the 60 steps recorded in Room 1 are not equivalent to the 83 steps from DFS, which achieved full coverage.
- **Turns**: **Inwards Spiral** had the lowest number of turns needed to complete coverage, followed by **DFS**, **Code**$_2$, and finally **Code**$_1$. The high number of turns in **Code**$_2$ explains its strategy, which involves frequent rotations to check for available space before moving.

This analysis clearly shows that **DFS** and **Inwards Spiral** were the fastest and most efficient approaches in terms of coverage and analysis time while **Code**$_2$ demonstrated a balance between effort and coverage, and **Code**$_1$ had the worst overall performance in all categories.

It is important to note that, although both **Code**$_2$ and **Code**$_1$ had worse analysis times than **DFS** and **Inwards Spiral**, their algorithm had an important factor that increased their assessment: `flipCoin()`.

The `flipCoin()` is a ROBO method that has a chance of 50% to return either true or false. Each time `flipCoin()` is called in RoboMind, the number of states in the internal representation for the program used by FDR duplicates. For example, in the RoboMind environment, when the robot collides with a wall and decides to either go north or west, it will *flip* *a coin*. If it is true, then it goes north. If it is false, then it goes west. For the state-machine that represents the CSP$_M$ specification inside, FDR considers two independent paths: one for the true evaluation and the other for the false evaluation. Thus, the state-machine size doubles after the `flipCoin()` call, as FDR looks for all possible outcomes. That is why **Code**$_2$ has a lower analysis time than **Code**$_1$ as it only *flips a coin* when it needs to. There is no point in *flipping a coin* to decide between north and south when the north is blocked, like **Code**$_1$ does. With a reduced state-machine, **Code**$_2$ shows to be faster than **Code**$_1$.

4.6.1.2 Optimal Local Coverage

The following analysis examines the performance of **Code**$_1$ and **Code**$_2$ on maps where the verification timed out the maximum one-hour waiting time. Since FDR does not provide intermediate results upon reaching a timeout, we set the maximum observable coverage within a 1-hour limit using the binary search approach described in Algorithm 1, adjusting the targeted coverage threshold in the refinement process.

By employing Algorithm 1, we identified the closest achievable coverage for each algorithm within a defined timeout, though this method extended the evaluation duration, as each algorithm potentially underwent up to seven executions, each capped at a maximum of one hour.

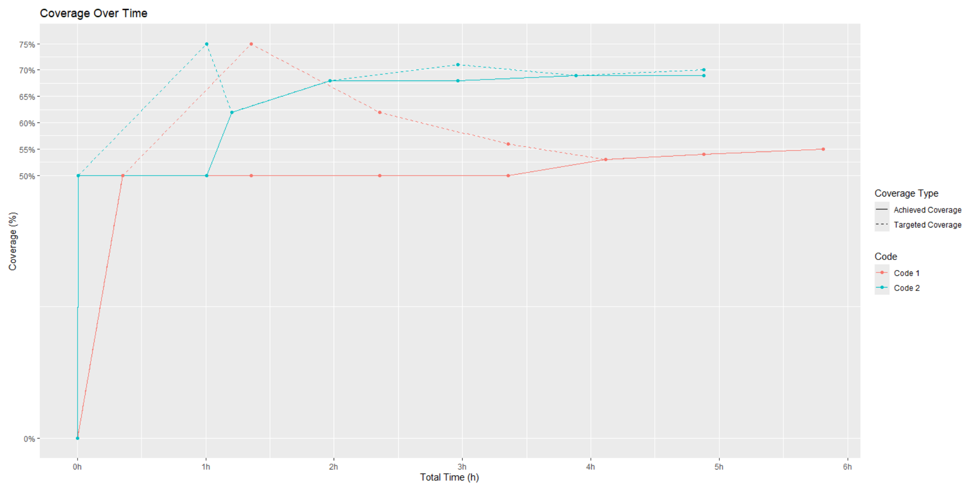
Figure 13 presents a line graph showing the total time required for each algorithm to reach maximum coverage within the Living Room map. Solid lines indicate each algorithm's peak coverage over time, while dashed lines mark target coverages used during binary search. Each line dot represents the target coverage and the maximum achieved by each algorithm. Notably, **Code**$_2$ reaches higher coverage faster than **Code**$_1$, achieving its peak within approximately 5 hours.

Table 5 details the highest coverage reached by **Code**$_1$ and **Code**$_2$ within the time constraints. Each performance metric reflects the highest coverage achieved by each code.

Table 5 – Optimal local coverage overview for single robots

Map	**Code**$_1$			**Code**$_2$		
	Effort	Turns	%	Effort	Turns	%
Living Room (19×12)	53	46	54	63	78	69
Room 2 (14×11)	58	43	60			

Figure 13 – Binary Search for the Living Room map
Source: Author (2025)



On the Room 2 map, **Code**$_2$ attained 100% coverage (as shown in Table 5) outperforming **Code**$_1$, which achieved 60%—a result below its potential maximum coverage. This outcome demonstrates **Code**$_2$ has a superior performance in this environment.

Similarly, in the Living Room, under identical time constraints, **Code**$_2$ consistently outperformed **Code**$_1$, achieving 69% coverage in contrast to **Code**$_1$'s 54%. Although these coverages do not represent the highest achievable by each algorithm without time restrictions, they reflect the maximum attainable within the given constraints, affirming **Code**$_2$ has the advantage.

4.6.2 Robot Size and Path Feasibility

In this section, we present and analyze the performance results from our approach comparing **Code**$_1$, **Code**$_2$, **Depth-First Search** and **Inwards Spiral**, in a more feasible grid, presented in Figure 8, with the robot occupying 3×3 cells. Table 6 presents the performance metrics for all four algorithms. As creating the map is a manual process, where we have to type each cell in a txt file from scratch by hand with the help of a figure, only one map is presented for this analysis.

Comparing the overall result, **Code**$_1$ failed to achieve 100% coverage while the other algorithms attained 100% value. **Inwards Spiral** showed the lowest effort and turn. **Code**$_2$ showed low effort but a high number of turns. All algorithms showed a higher analysis time than without increased radius, still showing that **DFS** and **Inwards Spiral** were the fastest in analysis time and lowest in turns. **Code**$_2$ and **Inwards Spiral** showed to possess the lowest effort. Even with a higher path feasibility with the robot occupying more cells, **Code**$_1$ showed

Table 6 – Optimal global coverage overview for robot size

Map	**Code$_1$**				**Code$_2$**			
	Time (s)	Effort	Turns	%	Time (s)	Effort	Turns	%
Room 1 (32×32)	–	–	–	–	5.5 ± 0.1	207	144	100%
Map	**Depth-First Search**				**Inwards Spiral**			
	Time (s)	Effort	Turns	%	Time (s)	Effort	Turns	%
Room 1 (32×32)	1.1 ± 0.1	349	24	100%	0.8 ± 0.1	200	18	100%

an inability to finish validation within the time constraint. Analyzing the highest possible coverage with Algorithm 1 in **Code$_1$** is unnecessary for this work, as it showed to reach 69% without path feasibility, so the highest possible coverage for **Code$_1$** will be not much distant of this value, as shown in Figure 9.

4.6.3 Discussion

Table 7 provides a high-level comparison of the four evaluated CPP algorithms based on analysis time, effort, turns, and achieved coverage. Each algorithm exhibits different trade-offs between efficiency and completeness, highlighting the impact of structured versus randomized decision-making in autonomous cleaning.

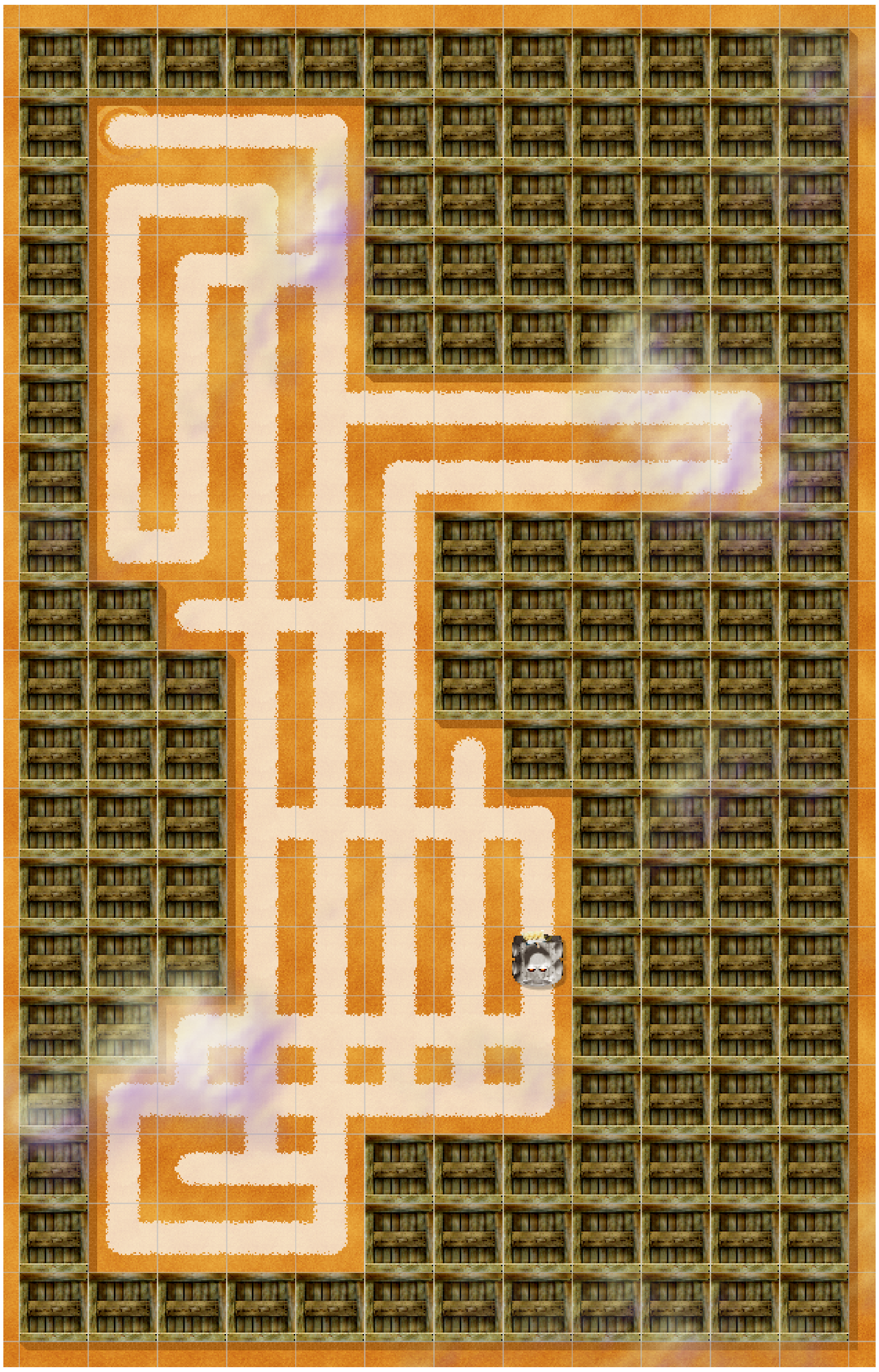
Table 7 – Summary table

Algorithm	**Time (s)**	**Effort**	**Turns**	**Coverage (%)**
Code$_1$	High	Moderate	Moderate	Inconsistent
Code$_2$	Moderate	Low	High	100%
DFS	Low	High	Low	100%
Inwards Spiral	Low	Moderate	Low	100%

Code$_1$, which follows a random-walk-based approach, demonstrates high analysis time and moderate analysis effort due to its reliance on stochastic movement. Although it has a moderate number of turns, its coverage remains inconsistent across different maps, as the lack of a structured strategy often leads to inefficient trajectories and redundant revisits to previously cleaned areas.

Code$_2$, while improving on **Code$_1$** by introducing a structured decision-making process, still exhibits moderate analysis time due to calls for `flipCoin()`. The increased turning frequency results from its strategy to prioritize determinism by reducing the number of states from the state-machine FDR generates while ensuring that it achieves 100% in the analyzed maps. While the FDR validation did not show **Code**$_2$ achieving 100% in the Living Room due to the time constraint, it demonstrates a potential of achieving 100% in the RoboMind simulation, as shown in Figure 14. However, the high turn rate contributes to lower overall efficiency, as the robot frequently reorients itself instead of prioritizing longer straight-line movements.

Figure 14 – Code$_2$ completeness in the Living Room
Source: Author (2025)



DFS, as expected from its structured exploration approach, achieves low analysis time and high effort as it thoroughly explores an area before backtracking. The low number of turns reflects the algorithm's preference for deep traversal before changing direction, allowing efficient movement with minimal reorientation overhead. Additionally, **DFS** consistently achieves full coverage, ensuring that all accessible areas are to be cleaned. However, its high effort cost means it covers more distance than necessary.

Finally, **Inwards Spiral** emerges as the most balanced strategy, achieving low analysis time, moderate effort, and low turns while consistently reaching 100%. The structured nature of this approach minimizes redundant movements and unnecessary turns, allowing the robot to navigate the environment with optimized path planning. Compared to **DFS**, it requires less effort to achieve full coverage, making it a highly efficient choice for structured cleaning tasks.

These results highlight the trade-offs between randomized and deterministic strategies and the impact of turn frequency on execution efficiency. While **Code**$_1$ struggles with inconsistency, **Code**$_2$ suffers from excessive turns. In contrast, **DFS** and **Inwards Spiral** ensure 100% coverage, with **Inwards Spiral** emerging as the most efficient algorithm overall.

Comparing the overall result, **Code**$_1$ failed to achieve 100% coverage while the other algorithms attained 100% value. **Inwards Spiral** showed the lowest effort and turn. **Code**$_2$ showed low effort but a high number of turns. **DFS** and **Inwards Spiral** were the fastest in analysis time and lowest in turns. **Code**$_2$ and **Inwards Spiral** showed to possess the lowest effort. Even with a higher path feasibility with the robot occupying more cells, **Code**$_1$ showed an inability to finish validation within the time constraint. If the amount of turns of **Code**$_2$ could be reduced in the RoboMind environment, it can be a strong candidate for an efficient CPP algorithm, even with its nondeterministic approach. To continue with this line of research, **Inwards Spiral** appears to be the most promising algorithm for efficient and complete coverage in structured environments, while **DFS** remains a reliable choice for thorough exploration despite its higher effort cost.

5 RELATED WORK

Galceran and Carreras (GALCERAN; CARRERAS, 2013) review several map representation techniques for robotic path planning, including grid-based, graph-based, and landmark-based approaches, providing a comprehensive analysis of CPP for mobile robots. These methods are widely used across robotic applications requiring efficient navigation. Our study differs by incorporating formal verification to assess the cleaning performance of the algorithms — an aspect not addressed in their research.

Pak et al. (PAK et al., 2022) applied grid-based path planning in agricultural robotics, focusing on mapping and localization to enable autonomous farming tasks. While we also focus on structured environments, our approach emphasizes formal validation, allowing for a systematic analysis of robot behavior across different scenarios.

Lin et al. (LIN et al., 2022) examined path-planning algorithms in multi-robot systems, analyzing centralized and decentralized decision-making for real-time applications. In contrast, our work focuses on the formal evaluation of individual trajectories before extending to multi-robot scenarios.

The chaotic motion planning approach in (AHURAKA et al., 2023) addresses the NP-hard problem of achieving full coverage while avoiding collisions, which aligns with our goal of optimizing AVCs in cluttered environments. While their approach maximizes exploratory coverage, it does not provide the guarantees of a formal analysis — something our CSP-based approach does.

Similarly, Govindaraju (GOVINDARAJU et al., 2023) proposed an optimized offline CPP algorithm for multi-robot systems, focusing on improving coverage efficiency in structured fields. While we share the same goal of optimizing coverage and effort, our study differentiates itself by using model-checking to verify the strategy before real-world implementation.

Dhaniya et al. (DHANIYA; UMAMAHESWARI, 2021) identified limitations in traditional path-planning algorithms for grid-based environments, particularly regarding robot size and corner navigation inefficiencies, and evaluated heuristics to minimize turns and search time. Our study addresses these challenges through precise formalization and CSP-based refinement analysis.

Thus, while existing works provide valuable insights into CPP, our study contributes to the field by integrating model-checking as a core tool for formally validating autonomous cleaning algorithms, ensuring performance optimization and verifiable correctness.

A deterministic execution environment is preferable primarily because formal verification relies on precisely defined states and transitions. Stochastic methods introduce probabilistic behavior, which, while useful for robustness testing, makes exhaustive verification significantly more difficult. A purely statistical approach may reveal trends, but it does not formally guarantee system properties such as deadlock freedom, liveliness, and worst-case execution bounds—which formal methods like CSP can provide. However, this does not mean that stochastic methods are ignored; the deterministic verification serves as a foundation, ensuring that core behaviors function correctly before introducing stochastic elements. Thus, while the current approach focuses on deterministic correctness, it does not preclude the possibility of integrating probabilistic elements.

CSP has a steeper learning curve than graphical modeling tools such as Petri Nets and Timed Automata, which offer visual representations of system behavior. Graphical tools provide an intuitive way to model concurrency and can often be easier to understand initially but may lack the expressiveness needed for complex concurrent interactions. CSP, while more abstract, offers distinct advantages, such as process composition, allowing hierarchical modeling and code reuse, which is difficult in purely graphical approaches; CSP's formal semantics prevent ambiguity, which is crucial for verifying correctness in concurrent systems, making it more reliable for verifying concurrent systems, and its algebraic nature allows for modular modeling, making it easier to reason about large and complex systems compared to purely visual approaches.

6 THREATS TO VALIDITY

As internal threats, we have:

- **Algorithm selection bias**: this paper is restricted to four relatively simple algorithms. The selection of **Code**$_1$ and **Code**$_2$ was justified as these algorithms represent a progression from basic to advanced approaches, enabling meaningful comparisons across common path planning strategies for AVCs;
- **Algorithm limitations**: while our 2D grid lacks full real-world complexity, it allows for controlled, systematic evaluation of core algorithm behaviors. Future enhancements in the approach and physical experiments will address real-world factors;
- **Extensibility Beyond Current Algorithms**: while this study evaluates four algorithms (**Code**$_1$, **Code**$_2$, **Depth-First Search**, and **Inwards Spiral**) within the RoboMind framework, the proposed methodology can be extended to analyze more sophisticated CPP strategies. However, the ROBO language imposes intrinsic limitations that prevent direct implementation of algorithms that require complex data structures or dynamic memory management, such as Breadth-First Search (BFS), A*, or deep learning approaches. The primary constraint is that ROBO lacks support for storing and manipulating structured data, such as queues, stacks, and priority lists, essential for many state-of-the-art planning algorithms. To overcome these limitations, an alternative approach would involve abstracting the high-level behavior of these advanced algorithms into CSP models rather than implementing them directly in ROBO. Future validations will define CSP specifications that capture the fundamental decision-making patterns of algorithms like BFS or A*, making it possible to analyze their efficiency and correctness within the FDR model checker;
- **State-space explosion and scalability**: FDR performs explicit-state refinement checking, which builds the complete state machine before asserting properties, which can lead to an exponential increase of validation time and resources as the robot program and map complexity grow. Time and memory can grow exponentially, limiting feasible model sizes.

As external threats, we have:

- **Generalizability**: this study focuses on a specific robot configuration and environment, which leads to bias; adapting other AVC algorithms and/or maps with appropriate modifications may demonstrate different results. Thus, it is not possible to generalize the presented results without more comprehensive evaluations;
- **Simulation fidelity**: simulations may not fully capture real-world complexities but provide a basis for initial evaluation. Planned physical experiments will validate results and explore the effects of factors like sensor noise and battery constraints;
- **Measurement limitations**: coverage percentage alone may not fully reflect cleaning performance. Future evaluations will incorporate metrics like energy consumption and cleaning quality; nonetheless, coverage remains a core algorithm behavior analysis baseline.

As construct threats, we have:

- **Operationalization of coverage:** coverage percentage is used as a proxy for cleaning quality, which may not capture dust removal effectiveness or uneven dirt distribution. To justify coverage as a baseline metric, we have to add secondary measures (revisits, turns, path length), and perform sanity checks on hand-crafted scenarios.
- **Abstraction mismatch (ROBO→CSP and grid world):** discretization (cell size), sensing/action model, and translation rules may omit aspects relevant to real robots. To provide a formal mapping from ROBO to CSP and validate with trace equivalence on test programs, we have to vary cell size and robot radius in experiments to assess robustness.
- **Map representativeness:** chosen maps may not represent realistic households. To document the selection criteria, we will have to include diverse layouts (corridors, open spaces, obstacles), and release all maps for reuse.

As conclusion threats, we have:

- **Statistical power and variance:** a small number of maps or runs can lead to unstable conclusions. To mitigate this, we will have to use multiple runs for stochastic algorithms, report mean and confidence intervals, and prefer effect sizes over sole point estimates.

- **Scope of claims:** causal statements beyond the controlled setting are unwarranted. We restrict conclusions to the defined grid abstraction and the tested configurations.

7 CONCLUSION

This study leverages CSP and the FDR model checker to design and verify the behavior of AVCs, focusing on CPP with 2D grid-based maps. CPP is essential in robotics, especially for autonomous cleaners, as it involves finding a path that covers all grid cells while avoiding obstacles.

This research demonstrates a structured translation method that could serve as a basis for further development in other robotic programming paradigms. While the current methodology is grounded in structured transformations of ROBO into CSP$_M$ using Spoofax, SDF3, and Stratego, alternative approaches could, for example, utilize Petri Nets to represent the model visually.

While other domain-specific language transformation tools exist, the methodology presented here is designed specifically for robotic algorithms. It ensures the formal correctness and verification efficiency of a ROBO program by translating it into CSP instead of execution efficiency. The core methodology of structured translation demonstrated here can be ported to other transformation frameworks, providing the groundwork for further adaptations where other robotic domain-specific languages can benefit from CSP-based verification.

We use a grid-based map to facilitate CPP, where each cell represents free spaces or an obstacle. This research introduces a formal approach for modeling and verifying AVC algorithms using CSP and RoboMind. Our strategy systematically evaluates algorithm performance regarding coverage, effort, and turns. By integrating CSP with RoboMind, we provide a verification model suited to AVCs and characterize its scalability limits. In our evaluation, it scales to moderate-size grid maps and program complexity; we also report time/memory bounds and timeouts, and outline mitigation strategies. Through comparative analysis of four algorithms, our strategy highlights coverage, efficiency, and computational cost differences. This approach is consistent with broader trends in robotics research, where formal verification is increasingly adopted as a foundational step in system design. Establishing logical correctness before integrating probabilistic elements or empirical models enhances the robustness of autonomous systems across domains.

The results indicate that randomized strategies, such as **Code**$_1$, lead to inconsistent coverage and higher analysis times. In contrast, structured approaches, such as Depth-First Search (DFS) and Inwards Spiral, consistently achieved full coverage with optimized efficiency. **Code**$_2$, an improved version of **Code**$_1$, managed to reach 100% coverage in all tested maps while maintaining lower effort than **DFS**. However, its frequent turning increased the analysis time, making it less efficient than **Inwards Spiral**. Among the evaluated algorithms, **Inwards Spiral** demonstrated the best balance between analysis time, effort, and number of turns, suggesting that it is the most effective approach for structured environments.

Limitations of the approach include simplified environmental models and idealized sensor data. Future work aims to:

- Increase model complexity for real-world accuracy;
- Extend the proposed methodology to more complex CPP algorithms;
- Incorporate additional layouts or randomly generated environments;
- Consider dynamic obstacles and room dimensions;
- Incorporate unconventional rooms and obstacle positions to assess adaptability;
- Conduct physical experiments to validate simulations;
- Implement real-time decision-making;
- Explore machine-learning techniques;
- Explore energy-aware path planning;
- Optimize system energy efficiency;
- Incorporate hybrid approaches, where CSP verification is complemented by empirical validation in more sophisticated simulators or hardware prototypes.

REFERENCES

AHURAKA, F.; MCNAMEE, P.; WANG, Q.; AHMADABADI, Z. N.; HUDACK, J. Chaotic Motion Planning for Mobile Robots: Progress, Challenges, and Opportunities. *IEEE Access*, v. 11, p. 134917–134939, 2023.

ARAÚJO, R.; MOTA, A.; NOGUEIRA, S. Analyzing Cleaning Robots Using Probabilistic Model Checking. In: BOUABANA-TEBIBEL, T.; BOUZAR-BENLABIOD, L.; RUBIN, S. H. (Ed.). *Theory and Application of Reuse, Integration, and Data Science*. Cham: Springer International Publishing, 2019. p. 23–51. ISBN 978-3-319-98056-0. Disponível em: <https://doi.org/10.1007/978-3-319-98056-0_2>.

BRUNSKILL, E. et al. Co-verification of industrial robotic systems using CSP and Simulink. *arXiv preprint*, 2021. Disponível em: <https://uia.brage.unit.no/uia-xmlui/handle/11250/2988388>.

CAVALCANTI, A. e. a. Modelling and verification for swarm robotics. *University of York Technical Report*, 2018. Disponível em: <https://www-users.york.ac.uk/~alcc500/publications/papers/CMSLRT18.pdf>.

CORPORATION iRobot. *Roomba® Robot Vacuum Cleaners*. 2025. <https://www.irobot.com/en_US/roomba.html>.

CORREIA, L. F. P. d. G. B.S. thesis, *Verificação eficiente de robôs educacionais*. 2021. Disponível em: <https://repository.ufrpe.br/handle/123456789/3975>.

DHANIYA, R. D.; UMAMAHESWARI, K. M. Critical Comparative Study of Robot Path Planning in Grid-Based Environment. In: *Journal of Physics: Conference Series*. [S.l.: s.n.], 2021. v. 1804, n. 1, p. 012193.

DOE, J.; SMITH, J. *Modelling the Turtle Python library in CSP*. 2022. <https://arxiv.org/pdf/2207.09706>.

ESCHMANN, H.; EBEL, H.; EBERHARD, P. Exploration-exploitation-based trajectory tracking of mobile robots using Gaussian processes and model predictive control. *Robotica*, p. 1–19, 2023.

FILHO, M. S. C.; MARINHO, R.; MOTA, A.; WOODCOCK, J. Analysing RoboChart with Probabilities. In: MASSONI, T.; MOUSAVI, M. R. (Ed.). *Formal Methods: Foundations and Applications*. Cham: Springer International Publishing, 2018. p. 198–214. ISBN 978-3-030-03044-5. Disponível em: <https://doi.org/10.1007/978-3-030-03044-5_13>.

Formal Systems (Europe) Ltd. *FDR4 Manual*. [S.l.], 2024. Accessed: July 2025. Disponível em: <https://dl.cocotec.io/fdr/fdr-manual.pdf>.

FOUNDATION, O. S. R. *Gazebo Simulator*. 2024. <https://gazebosim.org>.

GALCERAN, E.; CARRERAS, M. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, v. 61, n. 12, p. 1258–1276, 2013. ISSN 0921-8890. Disponível em: <https://www.sciencedirect.com/science/article/pii/S092188901300167X>.

GHASSEMI, F.; TRIPAKIS, S. *Formal Methods for CPS: State of the Art and Future Directions*. 2020. <https://par.nsf.gov/servlets/purl/10505257>.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. FDR3 — A Modern Refinement Checker for CSP. In: ÁBRAHáM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2014. (Lecture Notes in Computer Science, v. 8413), p. 187–201.

GOVINDARAJU, M.; FONTANELLI, D.; KUMAR, S. S.; PILLAI, A. S. Optimized Offline-Coverage Path Planning Algorithm for Multi-Robot for Weeding in Paddy Fields. *IEEE Access*, v. 11, p. 109868–109884, 2023.

HOARE, C. A. R. *Communicating sequential processes*. [S.l.: s.n.], 1978. v. 21. 666–677 p.

KALLEBERG, K. T. Stratego. *Crossroads*, ACM, v. 12, n. 3, p. 4–4, May 2006. ISSN 1528-4972. Disponível em: <http://dx.doi.org/10.1145/1144366.1144370>.

KATS, L. C.; VISSER, E. The Spoofax language workbench. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, ACM Press, 2010. Disponível em: <http://dx.doi.org/10.1145/1869542.1869592>.

LIN, S.; LIU, A.; WANG, J.; KONG, X. A Review of Path-Planning Approaches for Multiple Mobile Robots. *Machines*, v. 10, n. 9, 2022. ISSN 2075-1702. Disponível em: <https://www.mdpi.com/2075-1702/10/9/773>.

LUCKCUCK, M.; FARRELL, M.; FISHER, M. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *Autonomous Robots*, 2021. Disponível em: <https://mural.maynoothuniversity.ie/id/eprint/17468/1/MattLuckcuckAuto2021.pdf>.

MECK, L.; WAGNER, S. Fundamentals of Robotics Verification. *Embedded Systems Group*, 2018. Disponível em: <https://es.cs.rptu.de/publications/datarsg/Meck18.pdf>.

PAK, J.; KIM, J.; PARK, Y.; SON, H. I. Field Evaluation of Path-Planning Algorithms for Autonomous Mobile Robot in Smart Farms. *IEEE Access*, v. 10, p. 60253–60266, 2022.

RICKERT, M.; SIEVERLING, A.; BROCK, O. Balancing Exploration and Exploitation in Sampling-Based Motion Planning. *IEEE Transactions on Robotics*, v. 30, n. 6, p. 1305–1317, 2014.

RoboMind.net. *RoboMind.net – Educational Robotics Software*. 2006. <https://www.robomind.net>. Accessed: Sep. 1, 2025.

ROBOTICS, O. *Robot Operating System (ROS)*. 2024. <https://www.ros.org>.

ROSCOE, A. Modelling and verifying key-exchange protocols using CSP and FDR. In: *Proceedings The Eighth IEEE Computer Security Foundations Workshop*. [S.l.: s.n.], 1995. p. 98–107.

ROSCOE, A. *Understanding Concurrent Systems*. 1st. ed. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 184882257X. Disponível em: <https://doi.org/10.1007/978-1-84882-258-0>.

SCHNEIDERS, E.; KANSTRUP, A. M.; KJELDSKOV, J.; SKOV, M. B. Domestic Robots and the Dream of Automation: Understanding Human Interaction and Intervention. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2021. (CHI '21). ISBN 9781450380966. Disponível em: <https://doi.org/10.1145/3411764.3445629>.

UNKNOWN. *Room layout with bed, wardrobe and door*. Adapted from material available on Pinterest. Disponível em: <https://br.pinterest.com/pin/4362930883719929/>.

WANG, Z.; YUNSONG, L.; ZHANG, H.; LIU, C.; CHEN, Q. Sampling-Based Optimal Motion Planning With Smart Exploration and Exploitation. *IEEE-ASME Trans. on Mechatronics*, IEEE, v. 25, n. 5, p. 2376–2386, 2020.

WILTERDINK, R. *Design of a hard real-time, multi-threaded and CSP-capable execution framework*. 2011. Disponível em: <http://essay.utwente.nl/61066/>.