Bridging the Semantic Gap in *vGOAL* for Verifiable Autonomous Decision-Making

Yi Yang[a] and Tom Holvoet[b]
Imec-DistriNet, KU Leuven, 3001 Leuven, Belgium
{*yi.yang, tom.holvoet*}*@kuleuven.be*

Keywords: *vGOAL*, Semantic Gap, Interpreter, Model Checking, Autonomous Decision-Making.

Abstract: Verifiable autonomous decision-making requires bridging the semantic gap between the execution semantics of an agent programming language (APL) and the formal model used for verification. In this paper, we address this challenge for *vGOAL*, an APL derived from GOAL and designed for automated verification. We make three contributions. First, we identify the semantic gap in *vGOAL*: while both its interpreter and its model-checking framework implement the semantics of *vGOAL*, they differ in how they define the next program state for *vGOAL*. Second, we bridge the semantic gap by developing an improved interpreter for *vGOAL* that aligns with the model checker's formal semantics, thus ensuring correct verification results. Third, we introduce a stepwise refinement approach to address potential efficiency concerns arising from this semantic alignment. Through a case study in autonomous logistics, we demonstrate that while our approach introduces additional verification overhead, the efficient model-checking framework of *vGOAL* keeps this overhead manageable, making our solution practical for real-world applications.

1 INTRODUCTION

Verifiable autonomous decision-making requires bridging the semantic gap between the execution semantics of agent programming languages (APLs) and their formal models used for verification. This challenge is particularly evident in the relationship between APL interpreters, which execute programs, and their associated model-checking frameworks, which verify program properties.

Model-checking frameworks for APLs can be broadly categorized into two types. The first type is interpreter-based model-checking frameworks, e.g., the interpreter-based model checker (IMC) for GOAL (Hindriks, 2009), (Jongmans et al., 2010), (Jongmans, 2010). The second type is non-interpreter-based model-checking frameworks, e.g., the model-checking framework for AgentSpeak (Bordini and Hübner, 2005), (Bordini et al., 2003). When model-checking frameworks are developed upon the same interpreter as the running program, this approach establishes a solid foundation for semantic equivalence, as both operate under the same operational semantics. In this paper, we focus on *vGOAL*, an APL derived from GOAL with a specific emphasis on automated verification (Yang and Holvoet, 2023a). *vGOAL* is particularly advantageous due to its efficient model-checking framework, which is capable of validating complex autonomous systems involving multiple agents. For *vGOAL*, both its interpreter and its model-checking framework implement its operational semantics (Yang and Holvoet, 2023c), (Yang and Holvoet, 2024). However, they differ in their approach to defining the next program state for *vGOAL*, posing challenges for achieving a correct model for sound model-checking analyses.

To address this challenge, we make three main contributions. First, we identify and analyze the semantic gap between the *vGOAL* interpreter and its model-checking framework, providing a clear understanding of the discrepancies in their definitions of the next program state. Second, we develop an improved interpreter for *vGOAL* that aligns with the formal semantics of the model-checking framework, ensuring semantic consistency and correct verification results. Third, we introduce a stepwise refinement approach to address potential efficiency concerns arising from this semantic alignment. This approach mitigates the computational overhead introduced by the improved interpreter, ensuring that the solution remains practical for real-world applications.

[a] https://orcid.org/0000-0001-9565-1559
[b] https://orcid.org/0000-0003-1304-3467

Yang, Y. and Holvoet, T.
Bridging the Semantic Gap in vGOAL for Verifiable Autonomous Decision-Making.
DOI: 10.5220/0013262400003890

In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 1*, pages 580-587
ISBN: 978-989-758-737-5; ISSN: 2184-433X

Our approach effectively bridges the semantic gap in *vGOAL* while maintaining manageable computational costs, thanks to the efficiency of its model-checking framework. We demonstrate the practicality and effectiveness of our solution through a case study in autonomous logistics, involving three autonomous robots. This case study highlights how our approach enables reliable verification without compromising system performance.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 provides the preliminaries for *vGOAL*. Section 4 details the semantic gap problem in *vGOAL*. Section 5 presents our approach to bridging the semantic gap in *vGOAL* through an improved interpreter and discusses how stepwise refinement address potential performance overhead. Section 6 describes how to apply the stepwise refinement to a *vGOAL* program to improve the overall system efficiency through an autonomous logistic system. Finally, Section 7 concludes our paper.

2 RELATED WORK

Model checking is widely used in verifying both robotic systems and agent programs (Luckcuck et al., 2019). This section reviews key model-checking approaches for APLs, focusing on the semantic gap between program execution and verification.

Interpreter-based approaches use the APL interpreter directly in the model generation process, potentially offering better semantic alignment between program execution and the generated model for model checking. The MCAPL (Model-checking Agent Programming Languages) framework (Dennis, 2018) represents a significant advancement in this direction, supporting various APLs including Gwendolen, GOAL, SAAPL (Winikoff, 2007), ORWELL (Dastani et al., 2009), AgentSpeak, and 3APL (Hindriks et al., 1999). While MCAPL ensures close semantic alignment through its Agent Infrastructure Layer (AIL), it faces efficiency challenges (Dennis et al., 2012). Attempts to address these limitations through translation to more efficient model checkers like SPIN (Holzmann, 1997) and PRISM (Kwiatkowska et al., 2011) (Dennis et al., 2018) have shown promise but introduce new semantic gap concerns during the translation process.

The interpreter-based model checker (IMC) for GOAL (Jongmans et al., 2010), (Jongmans, 2010) takes a similar approach, using the program interpreter directly for state space generation. While this ensures semantic consistency, its limited state-space reduction capabilities and restriction to single-agent systems constrain its practical application.

These approaches typically translate agent programs into established model-checking languages. Early work with AgentSpeak (Bordini et al., 2003) used Promela for verification with SPIN, introducing semantic gaps through translation. Recent work (Yang and Holvoet, 2023b) demonstrates automated model checking for GOAL programs without interpreter dependency, though limited to single-agent systems.

Both approaches face challenges in bridging the semantic gap between program execution and model checking in APLs, with interpreter-based approaches generally offering better semantic alignment.

3 PRELIMINARIES: *vGOAL*

This section provides preliminaries to understand the semantic gap in *vGOAL*. We provide a concise overview of a *vGOAL* program, its operational semantics, and the shared implementation of its interpreter and its model-checking framework. For more details in *vGOAL*, its interpreter, and its model-checking framework, please refer to (Yang and Holvoet, 2023a), (Yang and Holvoet, 2023c), and (Yang and Holvoet, 2024), respectively.

Definition 1. *(vGOAL Program)*
A vGOAL program is defined as:

$$
\begin{aligned}
P &::= (MAS, RuleSets, Effects, Domain, Analyses),\\
MAS &::= Agent^{*},\\
Agent &::= (id, beliefs, goals, Msgs),\\
Msgs &::= sentMsgs, receivedMsgs,\\
RuleSets &::= Knowledge, Constraints, Actions, Sent, Events,\\
Analyses &::= Safety, Errors, FatalMsgs.
\end{aligned}
$$

A *vGOAL* program consists of the specification of the multi-agent system (*MAS*), five rule sets (*RuleSets*), the effects of actions ($Effects$), variable domains (*Domain*), and analyzed properties (*Analyses*). The syntax of *vGOAL* is based on first-order logic, with terms, predicates, and quantifiers. It imposes three key restrictions: finite domains for all variables, all variables must be quantified, and no negative recursion within each rule. These ensure a minimal model for each rule set.

Definition 2. *(Interpretation) The interpretation of beliefs and goals is defined as follows:*

$$
\begin{aligned}
beliefs &::= [b_1, ..., b_m]\\
I(beliefs) &::= \{b_1, ..., b_m\}\\
goals &::= [[g_{11}, ..., g_{1k}], ..., [g_n]]\\
I(goals) &::= \{a\text{-}goal\text{-}g_{11}, ..., a\text{-}goal\text{-}g_{1k}\}
\end{aligned}
$$

The interpretation of beliefs is a set of atoms. Notably, the interpretation of goals only pertains to the first goal of the agent. *a-goal* indicates the desired beliefs. The interpretation of goals is the key to merging state space of *vGOAL* programs, especially for multiple goals.

Definition 3. *(vGOAL States) A vGOAL state is defined as:*

$$state ::= \{subS_1, \ldots, subS_n\},$$
$$subS_i ::= id_i : (I(beliefs_i), I(goals_i)),$$

where $id_1, \ldots, id_n$ are unique identifiers for each substate.

A *vGOAL* state is composed of multiple substates. Each substate represents an agent, identified by a unique identifier, along with the interpretation of its beliefs and goals.

Definition 4. *(Operational Semantics of vGOAL) The operational semantics of* vGOAL *are defined by the transition:*

$$state \xrightarrow{Act} state',$$

where:

- $state, state' ::= subS_1, \ldots, subS_n$, *with* $n \geq 1$
- $subS_i ::= \{i : (I(beliefs_i), I(goals_i))\}$, *for* $i \in \{1, \ldots, n\}$
- $Act ::= events, actions$
- $events ::= \{id_1 : event_1, \ldots, id_n : event_n\}$, *with* $n \geq 1$
- $actions ::= \{id_1 : action_1, \ldots, id_n : action_n\}$, *with* $n \geq 1$
- id_i *are unique identifiers for each involved agent*

The operational semantics describe how each state transition is influenced by actions and events, which are specific to each agent.

For *vGOAL*, the implementations of its interpreter and its model-checking framework share the same implementation of the substate updates outlined in Algorithm 1. This algorithm implements all possible state transitions: $subS_i \xrightarrow{id_i:action_i, id_i:event_i,} subS_i'$, where substates evolve through two mechanisms: actions that modify agent beliefs, and events that can affect both beliefs and goals. This shared implementation is fundamental to ensure semantic consistency between program execution and the model checked by the model-checking framework, which we will explore in detail in the following sections.

Algorithm 1: Substate Update.

```
Function Expansion(state, P):
    foreach subS_i ∈ state do
        actions_i, events_i ← Reason(subS_i, P)
        subtrans_i ← ∅
        foreach action_i ∈ actions_i do
            subS'_i ← Update(subS_i, events_i, action_i)
            subT ← (subS, events_i, ation_i, subS'_i)
            subtrans_i ← subtrans_i ∪ {subT}
    Communicate among all agents
    subtransitions ← ⋃_1^n {(id_i : subtrans_i)}
    actions ← ⋃_1^n {(id_i : actions_i)}
return subtransitions, actions
```

4 SEMANTIC GAP

In this section, we analyze the fundamental difference between program executions in the *vGOAL* interpreter and its model-checking framework, illustrating this difference through a representative scenario.

Both implementations follow the same formal transition rule: $state \xrightarrow{Act} state'$. However, the model-checking framework waits for all agents to complete their actions before generating the next program state, while the interpreter updates the state as soon as one agent completes its action. This difference leads to distinct execution behaviors in practice. The interpreter's approach allows for more dynamic and responsive interactions with the environment, as it can adapt to changes as soon as they occur. In contrast, the model-checking framework considers all possible outcomes before transitioning to the next program state. These differences illustrate the trade-offs between real-time adaptability and exhaustive state exploration in multi-agent systems.

Algorithm 2 describes the stepwise execution of the *vGOAL* interpreter. It processes one state at a time and interacts with actual agents in the environment. It begins with a *vGOAL* program and real-time agent belief information as inputs. The execution starts with the initial state (s_0), based on agents' initial beliefs and goals. and sets the current state s and execution trace E to s_0. It then enters a loop that continues while any agent has remaining goals. The interpreter iteratively evaluates each agent's state to determine possible actions, allowing at most one action per substate. It identifies potential next substates by considering action outcomes and sends commands to agents with actions to perform. During the waiting phase, the interpreter updates each agent's substate using real-time information. The waiting phase ends as soon as one substate completes its action, after which the new state is set and the reasoning cycle begins again. This cycle continues until all agents have achieved their

Algorithm 2: Operational Semantics Implementation in the *vGOAL* Interpreter.

Input: a *vGOAL* program: P, real-time agent information: $Info$
Output: a program execution $s_0, (events_1, actions_1), s_1, \ldots, s_n$

```
// Initialization
s_0 ← {id_i : (I(beliefs_i), I(goals_i)) | i ∈ {1,...,n}}
s ← s_0, E ← s_0
// Continue until no gaols
while ∃i.I(goals_i) ≠ ∅ do
    subtrans, actions ← Expansion(s,P)
    foreach (id_i : actions_i) ∈ actions do
        Send to Agent_i: perform action_i.
    wait ← True, s ← ∅, t ← ∅
    while wait do
        foreach subS_i ∈ s do
            subS'_i ← Info
            subT ← (subS_i, _, action_i, subS'_i)
            t ← t ∪ {subT} s ← s' ∪ {subS'_i}
        foreach subT ∈ t do
            if subT ∈ subtrans_i then
                wait ← False,
                events ← events ∪ {id : event_i}
                actions ← actions ∪ {id : action_i}
                E ← E, (events, actions), s
return E
```

goals.

Algorithm 3 outlines the process of generating a transition system from a *vGOAL* program. The algorithm iteratively explores each state, generating possible transitions and updating the set of states and transitions accordingly. It captures the non-deterministic outcomes of agent actions by calculating all possible transitions from the current state, considering the Cartesian product of subtransitions for each agent. The loop continues until no new states are generated, at which point the algorithm terminates and returns the constructed transition system.

To illustrate the semantic gap in *vGOAL*, we present a representative multi-agent scenario depicted in Figure 1. The scenario involves two robots ($Robot_1$ and $Robot_2$) navigating through intermediate locations to reach a common destination ($Location_5$). $Robot_1$ starts from $Location_1$ and moves via $Location_2$, while $Robot_2$ begins at $Location_3$ and travels via $Location_4$. Access to locations is granted on a first-come-first-served basis, with $Robot_1$ having

Algorithm 3: Operational Semantics Implementation in the *vGOAL* Model-Checking Framework.

Input: a *vGOAL* program: P
Output: a transition system: (S, T, s_0, F)

```
// Initialization
s_0 ← {id_i : (I(beliefs_i), I(goals_i)) | i ∈ {1,...,n}}
S ← {s_0}, F ← ∅, T ← ∅
// Iterative state exploration
S_cur ← {s_0}
while S_cur ≠ ∅ do
    S_next ← ∅
    foreach s ∈ S_cur do
        subtrans, _, ← Expansion(s,P)
        transitions ← ∏_{i=1}^{n} subtrans_i,
          T ← T ∪ transitions
        foreach (s, _, _, s') ∈ transitions do
            S_next ← S_next ∪ {s'}
    // Terminal state identification
    foreach s ∈ S_next do
        if ∀i.I(goals_i) = ∅ then
            F ← F ∪ {s}
    // State updates
    S ← S ∪ S_next, S_next ← S_next \ F,
      S_cur ← S_next
return (S, T, s_0, F)
```

priority in simultaneous requests.

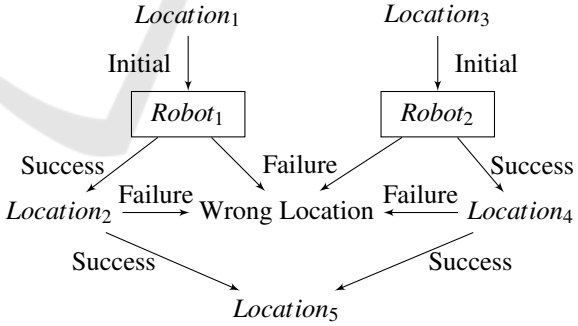

Figure 1: A Multi-Agent Scenario.

This scenario illustrates a fundamental difference between the interpreter and model-checking implementations. Initially, both implementations generate identical decisions, directing the robots toward their respective intermediate locations. However, their behaviors diverge in subsequent state transitions. The interpreter processes state changes asynchronously, updating the program state whenever either robot completes its movement. Consequently, the robot that reaches its intermediate location first gains access to

$Location_5$. In contrast, the model-checking framework processes state changes synchronously, waiting for both robots to complete their movements before transitioning to the next state. Under this implementation, $Robot_1$ consistently secures access to $Location_5$ due to its priority status.

5 BRIDGING SEMANTIC GAP

This section presents our approach to bridging the semantic gap in *vGOAL*. As discussed in Section 4, the *vGOAL* interpreter and model-checking framework differ in how they define the next program state. The interpreter updates the state as soon as one agent completes its action, while the model-checking framework waits for all agents to complete their actions. To address this gap, we propose two general approaches.

The first approach involves aligning both the *vGOAL* interpreter and the model-checking framework with the interpreter's definition of the next program state, where decisions are made as soon as any agent completes its action. This approach allows for more dynamic and responsive system behavior but exponentially increases the state space, thus worsening the state-space explosion problem in model checking.

The second approach aligns both components with the model-checking framework's definition, where decisions are made only after all agents complete their actions. Although this approach introduces potential efficiency concerns due to increased waiting times, it maintains a manageable state space and ensures consistency between the interpreter and model-checking framework.

Given that system efficiency can be improved through various software engineering techniques, we adopt the second approach and employ stepwise refinement of the original *vGOAL* program to mitigate the efficiency concerns. Our solution involves two sequential steps: (1) improving the *vGOAL* interpreter to ensure that autonomous decision-making occurs only after all agents have completed their current actions, and (2) applying stepwise refinement to the original *vGOAL* program to mitigate the increased execution time.

Algorithm 4 outlines the implementation of the operational semantics in the improved *vGOAL* interpreter. Compared with Algorithm 2, Algorithm 4 only revised the condition to generate the next decision for the autonomous system. Specifically, a new variable *terminate* is introduced in Algorithm 4 (see Line 13), which evaluates whether all agents complete their actions. This imposed restriction makes the implementation of the operational semantics in the *vGOAL* interpreter align with the implementation of the operational semantics in the model-checking framework for *vGOAL*.

Algorithm 4: Operational Semantics Implementation in the Improved *vGOAL* Interpreter.

```
   Input: a vGOAL program: P, real-time agent
          information: Info
   Output: a program execution
          s_0, (events_1, actions_1), s_1, ..., s_n
   // Initialization
 1 s_0 ← {id_i : (I(beliefs_i), I(goals_i)) | i ∈
     {1,...,n}}
 2 s ← s_0, E ← s_0
   // Continue until no gaols
 3 while ∃i.I(goals_i) ≠ ∅ do
 4     subtrans, actions ← Expansion(s,P)
 5     foreach (id_i : actions_i) ∈ actions do
 6         Send to Agent_i: perform action_i.
 7     wait ← True, s ← ∅, t ← ∅
 8     while wait do
 9         foreach subS_i ∈ s do
10             subS'_i ← Info
11             subT ← (subS_i, _, action_i, subS'_i)
12             t ← t ∪ {subT} s ← s' ∪ {subS'_i}
13         terminate ← True
14         foreach subT ∈ t do
15             if subT ∈ subtrans_i then
16                 events ← events ∪ {id : event_i}
17                 actions ← actions ∪ {id :
                    action_i}
18                 E ← E, (events, actions), s
19             else
20                 terminate ← False,
21             wait ← ¬terminate
22 return E
```

We acknowledge that the improved *vGOAL* interpreter may result in a long waiting time for some agents when each agent needs a different time to complete its action, which increases the overall execution time for the whole autonomous system.

However, we point out that this efficiency issue can be properly addressed if we introduce the stepwise refinement to the *vGOAL* program. Specifically, we can refine actions that may take a long time into multiple actions that take a shorter time, thus making the waiting time shorter, thereby improving the system efficiency. For example, consider a robot's movement action from location A to location D, which takes a long time to complete. Instead of having a single "move to location D" action, we can refine it

into multiple shorter actions that traverse through intermediate locations: ”move from A to B”, ”move B to C”, and ”move C to D”. This way, other agents can continue their tasks after each intermediate movement is completed, rather than waiting for the entire from A to D movement to finish. This refinement significantly reduces overall waiting time and improves system efficiency.

While stepwise refinement may increase the state space, we hypothesize that this growth can be controlled to remain linear rather than exponential. This hypothesis is based on the observation that the additional states represent intermediate steps of existing actions rather than entirely new behavioral branches. Furthermore, the verification of properties, including safety and liveness, remains unchanged despite the refinement of actions. Future experiments will be conducted to validate this hypothesis and quantify the impact of stepwise refinement on state-space growth.

6 CASE STUDY

This section presents a case study to illustrate how stepwise refinement addresses efficiency issues introduced by the improved *vGOAL* interpreter. The case study involves three autonomous robots operating in a logistic system. We conduct a comparative analysis to demonstrate the impact of stepwise refinement on system efficiency and empirically evaluate the time costs for model checking. All experiments were conducted on a MacBook Air 2020 with an Apple M1 and 16GB of RAM. The complete *vGOAL* specifications are available at (Yang, 2024).

The autonomous logistic system is expected to deliver three workpieces from one of the two pick stations to the delivery destination. The autonomous logistic system consists of three autonomous robots: $Robot_1$, $Robot_2$, and $Robot_3$. Each robot can perform four actions: *move*, *pick*, *drop*, and *charge*. Uncertainty happens in the execution of actions, i.e., each action can either succeed or fail, leading to a desired state or the system crash.

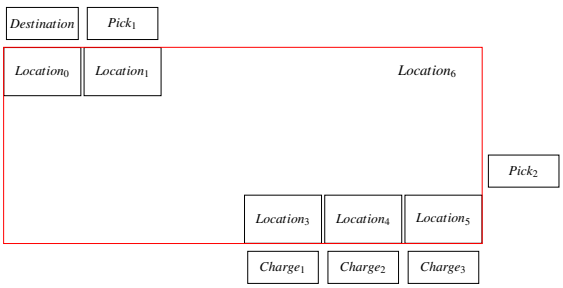

Figure 2: Environment before Refinement.

Now, we introduce the environment where the autonomous system operates before the stepwise refinement. Figure 2 presents the layout of the divided into eight areas, from $Location_0$ to $Location_7$. Eight areas include one delivery destination ($Location_0$); two pick stations ($Location_1$, $Location_2$); three charging stations ($Location_3$, $Location_4$, $Location_5$); and the rest area ($Location_6$). Initially, $Robot_1$ locates at $Location_3$; $Robot_2$ locates at $Location_4$; and $Robot_4$ locates at $Location_5$. Additionally, a maximum of one robot can stand on the locations from $Location_0$ to $Location_6$.

In this autonomous logistic system, each robot can only perform four actions. The execution time to perform *move* can vary a lot and this action needs location permission, while the execution time to perform the other three actions (*pick*, *drop*, and *charge*) are relatively stable, and do not require any critical resources. Hence, *move* is the key to affecting the overall execution time for this autonomous system. For example, the *vGOAL* interpreter initially makes a decision for $Robot_1$ and $Robot_2$: move to $Location_1$, and move to $Location_2$, respectively. $Robot_3$ has to wait to get the location permission for $Location_1$ until both $Robot_1$ and $Robot_2$ achieve their goals.

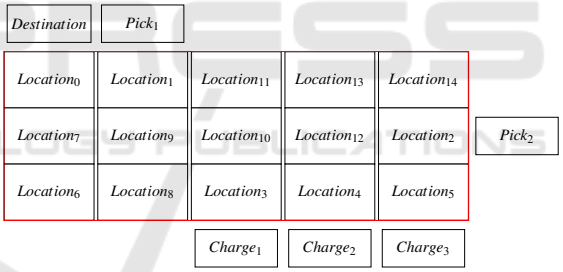

Figure 3: Environment after Refinement.

To address the inefficiency, we refined the environment to 15 areas as shown in Figure 3. Specifically, we keep the original six locations from $Location_0$ to $Location_5$, and we refine one original location ($Location_6$) to the current nine areas (from $Location_6$ to $Location_{14}$. In the refined environment, each execution time for the *move* action will be relatively stable, and robots do not have to wait to get the *move* command until other robots complete a long journey.

We illustrate the efficiency improvement between before and after the refinement. First, we use the improved *vGOAL* interpreter outlined in Algorithm 4 and the model-checking framework for *vGOAL*. The *vGOAL* program describes how the autonomous logistic system works in the environment as shown in Figure 2. Second, we use the improved *vGOAL* interpreter outlined in Algorithm 4 and the model-

checking framework for *vGOAL*. The *vGOAL* program describes how the autonomous logistic system works in the refined environment as shown in Figure 3.

Table 1: Stepwise Refinement in *vGOAL* Program.

Original Path	Refined Path	Time
$3 \rightarrow 1$	$3 \rightarrow 8 \rightarrow 9 \rightarrow 1$	$3t_1$
$4 \rightarrow 1$	$4 \rightarrow 8 \rightarrow 9 \rightarrow 1$	$3t_1$
$5 \rightarrow 1$	$5 \rightarrow 8 \rightarrow 9 \rightarrow 1$	$3t_1$
$3 \rightarrow 2$	$3 \rightarrow 10 \rightarrow 12 \rightarrow 2$	$3t_1$
$4 \rightarrow 2$	$4 \rightarrow 12 \rightarrow 2$	$2t_1$
$2 \rightarrow 0$	$2 \rightarrow 14 \rightarrow 13 \rightarrow 11 \rightarrow 1 \rightarrow 0$	$5t_1$
$0 \rightarrow 3$	$0 \rightarrow 7 \rightarrow 6 \rightarrow 8 \rightarrow 3$	$4t_1$
$0 \rightarrow 4$	$0 \rightarrow 7 \rightarrow 6 \rightarrow 8 \rightarrow 4$	$4t_1$
$0 \rightarrow 5$	$0 \rightarrow 7 \rightarrow 6 \rightarrow 8 \rightarrow 5$	$4t_1$
$1 \rightarrow 0$	$1 \rightarrow 0$	t_1
$5 \rightarrow 2$	$5 \rightarrow 2$	t_1

Table 1 illustrates the refinement of movement paths in our case study. Among the four possible robot actions, the *move* action exhibits significant variability in execution time, quantified using t_1. This variability is primarily influenced by the distance a robot must travel, leading to substantial waiting times for other robots and reducing overall system efficiency.

In our analysis, robots can traverse 11 different paths to achieve their delivery objectives. We identified nine paths that need to be refined, each requiring multiple t_1 units for a *move* action, while only two paths were efficient, taking approximately t_1. To optimize the system, we concentrated our refinement efforts on these nine longer paths. Our approach involved breaking down each long path into shorter segments, ensuring that each *move* action would take approximately t_1 to complete. This refinement approach reduces robot waiting times and improves overall system performance.

The total system execution time is calculated as the sum of sequential action steps, where multiple robots can operate simultaneously in each step. For example, when all three robots perform a charging action concurrently, the execution time for this step is simply t_2. Similarly, when $Robot_1$ moves from $Location_3$ to $Location_1$ (taking $3t_1$) while $Robot_2$ moves from $Location_4$ to $Location_2$ (taking $2t_1$), and $Robot_3$ remains idle, the execution time for this step is $max(3t_1, 2t_1, 0) = 3t_1$. This calculation method ensures an accurate representation of both sequential and parallel robot operations.

We ran both the original and refined *vGOAL* programs within a model-checking framework, generating the same longest execution that can be produced by the *vGOAL* interpreter. Table 2 shows the details of the execution time before and after refinement. The execution time before refinement includes $26t_1$ for seven unrefined *move* actions, t_2 for one *charge* action, $2t_3$ for two *pick* actions, and $3t_4$ for three *drop* actions. The execution time after refinement includes $17t_1$ for 17 refined *move* actions, t_2 for one *charge* action, $2t_3$ for two *pick* actions, and $2t_4$ for two *drop* actions.

Table 2: Execution Time before and after the Refinement.

Duration	Before Refinement	After Refinement
move	$26t_1$	$17t_1$
charge	t_2	t_2
pick	$2t_3$	$2t_3$
drop	$3t_4$	$2t_4$

The refinement of the *vGOAL* program significantly improved the efficiency of robot actions, particularly in reducing the time spent on *move* actions. By breaking down longer paths into shorter segments, we achieved a reduction in the total execution time from $26t_1$ to $17t_1$ for *move* actions. This refinement of *move* minimizes waiting times for other robots and improves the overall system performance.

Table 3: Model-Checking Before and After Refinement.

Indicators	Original	Refined
Number of States	1186	2635
Model-checking Time (s)	40.88	218.84

We recognize that the refinement process leads to an expansion of the state space, which in turn increases the time required for model-checking. Table 3 illustrates the model-checking results both before and after the refinement. In this case study, we focused on verifying the safety and liveness properties. As expected, the state space approximately doubled, resulting in an increase of about 160 seconds in model-checking time. Despite this increase, the additional time remains manageable and within acceptable limits. Despite this increase in computational overhead, we consider this trade-off acceptable for several reasons. First, the model-checking time remains within practical limits, adding only about three minutes to the verification process. Second, this one-time verification cost is outweighed by the long-term benefits of improved runtime efficiency in the actual robot system.

7 CONCLUSION AND FUTURE WORK

This paper addresses a critical challenge in verifiable autonomous decision-making: bridging the semantic gap between program execution and model checking in *vGOAL*. We have made three key contributions: (1) identifying and analyzing the semantic gap in *vGOAL*, (2) developing an improved interpreter that aligns the implementation of the next program state, and (3) demonstrating how stepwise refinement can effectively address potential efficiency issues.

Our case study of an autonomous logistics system with three mobile robots validates our approach. The results show that while our improved interpreter may introduce some execution overhead, the stepwise refinement successfully reduces the execution time for *move* actions by 34.6%. Although the refinement process increased the state space and model-checking time, the additional computational cost remained manageable, demonstrating the practicality of our approach.

Future work will focus on two directions: (1) extending experimental validation across a broader range of multi-agent scenarios and real-world environments, and (2) conducting comprehensive scalability analysis with increasing system complexity. These extensions will further validate and enhance our approach to developing reliable autonomous decision-making systems.

ACKNOWLEDGEMENTS

This research is partially funded by the Research Fund KU Leuven.

REFERENCES

Bordini, R. H., Fisher, M., Pardavila, C., and Wooldridge, M. (2003). Model checking AgentSpeak. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 409–416.

Bordini, R. H. and Hübner, J. F. (2005). BDI agent programming in AgentSpeak using Jason. In *International workshop on computational logic in multi-agent systems*, pages 143–164. Springer.

Dastani, M., Tinnemeier, N. A., and Meyer, J.-J. C. (2009). A programming language for normative multi-agent systems. In *Handbook of Research on Multi-Agent Systems: semantics and dynamics of organizational models*, pages 397–417. IGI Global.

Dennis, L. A. (2018). The mcapl framework including the agent infrastructure layer and agent java pathfinder. *The Journal of Open Source Software*.

Dennis, L. A., Fisher, M., and Webster, M. (2018). Two-stage agent program verification. *Journal of Logic and Computation*, 28(3):499–523.

Dennis, L. A., Fisher, M., Webster, M. P., and Bordini, R. H. (2012). Model checking agent programming languages. *Automated software engineering*, 19(1):5–63.

Hindriks, K. V. (2009). Programming rational agents in GOAL. In *Multi-agent programming*, pages 119–157. Springer, Berlin, Heidelberg.

Hindriks, K. V., De Boer, F. S., Van der Hoek, W., and Meyer, J.-J. C. (1999). Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2:357–401.

Holzmann, G. J. (1997). The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295.

Jongmans, S. (2010). Model checking GOAL agents.

Jongmans, S.-S. T., Hindriks, K. V., and Van Riemsdijk, M. B. (2010). Model checking agent programs by using the program interpreter. In *Computational Logic in Multi-Agent Systems: 11th International Workshop, CLIMA XI, Lisbon, Portugal, August 16-17, 2010. Proceedings 11*, pages 219–237. Springer.

Kwiatkowska, M., Norman, G., and Parker, D. (2011). Prism 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 585–591. Springer.

Luckcuck, M., Farrell, M., Dennis, L. A., Dixon, C., and Fisher, M. (2019). Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–41.

Winikoff, M. (2007). Implementing commitment-based interactions. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8.

Yang, Y. (2024). Supplementary Documents. https://github.com/yiyangvGOAL/vGOALICAART2025.git.

Yang, Y. and Holvoet, T. (2023a). *vGOAL*: a GOAL-based specification language for safe autonomous decision-making. In *Engineering Multi-Agent Systems: 11th International Workshop, EMAS 2023, London, UK, 29-30 May 2023, Revised Selected Papers*.

Yang, Y. and Holvoet, T. (2023b). Making model checking feasible for goal. *Annals of Mathematics and Artificial Intelligence*.

Yang, Y. and Holvoet, T. (2023c). Safe autonomous decision-making with vGOAL. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics. The PAAMS Collection*. Guimarães, Portugal.

Yang, Y. and Holvoet, T. (2024). Model Checking of vGOAL. *arXiv*, Preprint. Preprint.