vGOAL: a GOAL-based Specification Language for Safe Autonomous Decision-Making

Yi Yang[0000−0001−9565−1559] and Tom Holvoet[0000−0003−1304−3467]

imec-DistriNet, KU Leuven, 3001 Leuven, Belgium
{yi.yang,tom.holvoet}@kuleuven.be

Abstract. Formal verification is a reliable approach to addressing safety concerns in autonomous applications. We have designed *vGOAL* based on the internal logic of the GOAL agent programming language, which serves as the formal specification language of our innovative formal approach to safe autonomous decision-making. A detailed description of *vGOAL* is necessary to present and justify our approach to safe autonomous decision-making, yet it is currently missing. Therefore, this paper aims to provide a comprehensive description of *vGOAL*, including its formal syntax, its operational semantics, a real-world robotic application, and a comparison with several comparable agent programming languages, namely, GOAL, Gwendolen, and AgentSpeak (Jason).

Keywords: Formal Specification · Autonomous Decision-Making · Safety Assurance · *vGOAL*

1 Introduction

The applications of autonomous systems have seen a remarkable increase in recent years. These systems are capable of operating without human intervention to achieve complex goals. As autonomous applications become increasingly common in industries like manufacturing and transportation, it is crucial to ensure their safety.

Safe autonomous decision-making is one of the key challenges in developing autonomous robotic applications. Agent programming languages (APLs), including AgentSpeak [2], Jason [3], Gwendolen [8], and GOAL [10], have been extensively researched for programming autonomous agents for decades, indicating two facts: (1) A multi-agent system can properly model agent-based autonomous systems; (2) APLs are well-suited for tackling the challenge of the decision-making of agent-based autonomous systems. Despite the potential benefits of APLs in the development of autonomous robotic applications, their research has not been widely used in the field. Integration with the Robot Operating System (ROS) may expand their applications to robotics, as ROS has become the de facto standard for developing robotic applications. If an APL has built-in support for ROS, it would be advantageous to integrate it with ROS-based robotic applications.

The Belief-Desire-Intention (BDI) model is a popular reasoning mechanism utilized in various APLs including Jason and Gwendolen [4]. GOAL shares many features with BDI APLs, such as beliefs and goals, but it is primarily a rule-based APL that differs in its approach to action selection [4]. Specifically, while BDI APLs select actions from a plan library, GOAL derives actions based on its rules to fulfill goals, making it highly suitable for specifying autonomous decision-making.

To facilitate safe decision-making of agent-based autonomous systems, we have developed *vGOAL*, which is a GOAL-based specification language that focuses exclusively on the internal logic reasoning mechanism of GOAL, motivated by three primary considerations. First, the decision-making mechanism of GOAL is highly suitable for autonomous decision-making, but many of its specifications are irrelevant to this domain, such as environment specifications. Second, the intrinsic logic-based nature of GOAL makes it highly suitable for formal verification, which is ideal for providing safety assurance for autonomous decision-making. Third, GOAL cannot directly access ROS, which limits its applicability in robotic applications. Therefore, *vGOAL* can be highly valuable for safe autonomous decision-making used in robotic applications, as it can leverage the strengths of GOAL, ROS, and formal verification.

On the basis of *vGOAL*, we have developed a three-stage formal approach to safe autonomous decision-making: formal specification using *vGOAL*, safe decision generation using the *vGOAL* interpreter, and the verification of *vGOAL* using an automated translator for *vGOAL* and a PCTL model checker (Storm [6] or PRISM [15]). Additionally, we have integrated the *vGOAL* interpreter into ROS via rosbridge to facilitate implementation and execution. We validated our approach in a real-world autonomous logistic system consisting of three autonomous mobile robots. There are three demonstration videos accessible for viewing at [18].

In [19], we established the preliminary groundwork for the formal specification and verification of *vGOAL* by outlining how to verify a GOAL program with specific restrictions, including a stratified program, a single agent, and a single goal. Building on this initial work, we described the rationale and implementation of the three-stage formal approach in [20]. [17] presents a high-level overview of the three-stage formal approach. However, a detailed description of *vGOAL* is crucial to thoroughly describing our approach to safe autonomous decision-making, similar to the descriptions of Gwendolen in [7] and in [8], and of GOAL in [10]. Therefore, the purpose of this paper is to provide a detailed explanation of *vGOAL*.

The paper is structured as follows. In Section 2, we present the formal syntax of *vGOAL*. In Section 3, we present the operational semantics of *vGOAL*. In Section 4, we demonstrate how to use *vGOAL* with a validated real-world autonomous logistic system. In Section 5, we will discuss the essential features of *vGOAL* and provide a comparative analysis with other APLs, namely GOAL, Gwendolen, and AgentSpeak (Jason). In Section 6, we draw conclusions on *vGOAL*.

2 Formal Syntax

In this section, we introduce the formal syntax of *vGOAL*. Initially, we introduce the core elements of *vGOAL*, highlighting its fundamental basis in first-order logic. Next, we delve into the predefined functions and the rule construction within vGOAL, elucidating its constraints within the first-order logic framework. Finally, we present the high-level components of *vGOAL* specifications.

The first part of the *vGOAL* syntax, which includes elements like terms and predicates, conforms to the conventions established in first-order logic. Nevertheless, in contrast to first-order logic, the specification allows for the optional indication of the domains associated with universally quantified variables. Minimal model generation is required for the generation of autonomous decision-making, hence necessitating the inclusion of domain specifications for universally quantified variables.

$$
\begin{array}{ll}
word & ::= char\ |num \\
constant & ::= word\ constant^* \\
variable & ::= char\ word^* \\
constant\_list & ::= constant\ constant^* \\
predicate\_name & ::= char\ word^* \\
ground\_atoms & ::= predicate\_name\ \text{'('}\ constant\_list\text{')'} \\
term & ::= constant \mid variable \\
term\_list & ::= term \mid term\ \text{','}\ term\_list \\
p & ::= predicate\_name\ (term\_list) \\
neg\_p & ::= \neg p \\
Uni\_Q & ::= \forall\ variable \mid \forall\ variable \in D \mid Uni\_Q^* \\
Ex\_Q & ::= \exists\ variable \mid Ex\_Q^*
\end{array}
$$

The second part of the *vGOAL* syntax involves some keywords, predefined functions, and the way of constructing rules.

R represents a group of agents, with its domain consisting of three distinct elements: *all*, *allother*, and *id*. Specifically, *all* and *allother* are keywords in *vGOAL*, denoting all agents and all agents within the multi-agent system excluding the individual responsible for transmitting messages, respectively; and *id* designates a particular agent.

For describing communication among agents, *vGOAL* offers six predefined functions: *send*:, *send*!, *send*?, *sent*:, *sent*!, and *sent*?. Like in GOAL, Message specifications in *vGOAL* differentiate among three types: indicative messages, indicated by the functions $send{:}(R, p)$ and $sent{:}(R, p)$; declarative messages, defined as $send!(R, p)$ and $sent!(R, p)$; and interrogative messages, represented as $send?(R, p)$ and $sent?(R, p)$. A sent message is represented as msg_s and encompasses three distinct elements within its domain: $send{:}(R, p)$, $send!(R, p)$, and $send?(R, p)$. Conversely, a received message is symbolized as msg_r and comprises three distinct elements within its domain: $sent{:}(R, p)$, $sent!(R, p)$, and

$sent?(R, p)$. Moreover, in a msg_s, R represents the recipients of the message, whereas in a msg_r, it signifies the sender.

Similarly to GOAL, *vGOAL* incorporates an event processing component responsible for handling communication messages and effecting changes in goals and beliefs. To facilitate these modifications, *vGOAL* provides four predefined functions: *insert*, *delete*, *adopt*, and *drop*. *response* signifies the outcome of event processing, which may encompass the generation of sent messages, the alteration of beliefs and goals, or both.

The minimal model serves as the foundation for establishing the semantics of *vGOAL*. Constructing the minimal model involves employing $qrule_i (1 \leq i \leq 6)$, which enforces three constraints on first-order logic: the quantification for each variable, a finite domain for each variable, and the absence of negative recursion. Furthermore, $qrule_7$ is used to define the effects of actions, and it is not used to deduce the subsequent safe autonomous decision.

$$
\begin{array}{ll}
id & ::= constant \\
R & ::= all | allother | id \\
msg_s & ::= send{:}(R, p) | send!(R, p) | send?(R, p) \\
msg_r & ::= sent{:}(R, p) | sent!(R, p) | sent?(R, p) \\
update & ::= insert(b) | delete(b) | adopt(g) | drop(g) \\
response & ::= msg_s | update \\
b & ::= ground\_atoms \\
g & ::= ground\_atoms \\
hs & ::= True | p \wedge hs | neg\_p \wedge hs \\
lh & ::= a\text{-}goal(p) \wedge hs \\
rule_1 & ::= hs \rightarrow p \\
qrule_1 & ::= Uni\_Q \ Ex\_Q \ rule_1 \\
rule_2 & ::= lh \rightarrow p \\
qrule_2 & ::= Uni\_Q \ Ex\_Q \ rule_2 \\
rule_3 & ::= hs \rightarrow msg_s \\
qrule_3 & ::= Uni\_Q \ Ex\_Q \ rule_3 \\
rule_4 & ::= msg_r \wedge hs \rightarrow response \\
qrule_4 & ::= Uni\_Q \ Ex\_Q \ rule_4 \\
rule_5 & ::= lh \rightarrow response \\
qrule_5 & ::= Uni\_Q \ Ex\_Q \ rule_5 \\
rule_6 & ::= hs \rightarrow response \\
qrule_6 & ::= Uni\_Q \ Ex\_Q \ rule_6 \\
rule_7 & ::= hs \rightarrow hs \\
qrule_7 & ::= Uni\_Q \ Ex\_Q \ rule_7
\end{array}
$$

The final part of the *vGOAL* syntax involves the high-level components of the *vGOAL* specification. The specification includes agent specifications and system specifications.

MAS denotes all agents' specifications involved in the multi-agent system. An agent specification consists of five essential components: a unique identifier: *id*, beliefs: B, goals: $goals$, sent messages: M_S, and received messages: M_R. The beliefs of an agent B consist of B_{sensor} and B_{prior}. B_{sensor} denotes the real-time beliefs obtained from sensors. B_{prior} denotes the prior beliefs that are essential for agents but cannot be received from sensors. An agent can have multiple goals, denoted by $goals$. Each goal (G) consists of goal bases.

System specifications in *vGOAL* involve six rule sets, each with a unique designation: K represents the knowledge base, C denotes enabled constraints, A refers to action generation, S pertains to sent message generation, P concerns event processing, and E describes action effects. Moreover, *a-goal* is a predefined function to evaluate if its argument is included in the first goal base.

$$
\begin{array}{ll}
Agent & ::= (id, B, goals, M_S, M_R) \\
MAS & ::= Agent^* \\
D & ::= constant\_list \\
B_{sensor} & ::= b^* \\
B_{prior} & ::= b^* \\
B & ::= B_{sensor}\ \ B_{prior} \\
G & ::= g^* \\
goals & ::= G^* \\
M_S & ::= msg_s^* \\
M_R & ::= msg_r^* \\
K & ::= qrule_1^*\ ground\_atom^* \\
C & ::= qrule_2^* \\
A & ::= qrule_1^* \\
S & ::= qrule_3^* \\
P & ::= qrule_4^*\ qrule_5^*\ qrule_6^* \\
E & ::= qrule_7^*
\end{array}
$$

Remark: Belief Base and Current Beliefs

In *vGOAL*, the belief base contains information that cannot be inferred by logical deduction. More specifically, an agent's current beliefs are obtained by combining its belief base with its knowledge base. As a result, the belief base represents a subset of an agent's current beliefs.

3 Operational Semantics

This section presents the operational semantics of $vGOAL$. Initially, we establish the semantics for high-level components in $vGOAL$. Subsequently, we explain how $vGOAL$ generates autonomous decisions through its reasoning cycle. The operational semantics of $vGOAL$ encompasses function updates and the generation of minimal models for first-order theories constrained by the $vGOAL$ syntax.

We use I to define the interpretations of high-level components in $vGOAL$. The principles to interpreters $vGOAL$ specifications are as follows.

- If $Spec ::= Agent, I(Spec) = id : (I(B), I(goals), I(M_S), I(M_R))$.
- If $Spec ::= e^*$, $I(Spec) = \bigcup I(e)$.
- If $Spec ::= e_1...e_n$, $I(Spec) = \bigcup^{n} I(e_i)$.
- If $Spec ::= goals$, and $goals ::= GG^*|Empty$, $I(Spec) = I(G)$ or $I(Spec) = \emptyset$.
- If $e ::= ground\_atom$, $I(e) = True$.
- The interpretation of logical operators, such as $\wedge$, $\neg$, and $\rightarrow$, adheres to the standard conventions of first-order logic.

Following the above interpretation principles, each high-level component of $vGOAL$ specifications, including B, G, $goals$, M_S, M_R, K, C, A, S, P, and E, is converted to a first-order theory constrained by $vGOAL$ syntax.

We use $I(Agent)$ and $I(MAS)$ as the foundation for constructing $vGOAL$'s substate and state, respectively. A substate represents an agent's state, while the state captures the autonomous system's state. A substate includes a unique identifier, beliefs, and goals, while the full information of the substate adds sent and received messages. We formally define a $vGOAL$ state and its corresponding information as follows:

$$\begin{aligned}
substate &::= id : (I(B), I(goals)),\\
sub_info &::= id : (I(B), I(goals), I(M_S), I(M_R)),\\
state &::= state \cup \{substate\}|\emptyset,\\
state_info &::= state_info \cup \{sub_info\}|\emptyset.
\end{aligned}$$

The core component of the reasoning cycle is the generation of the minimal model of a first-order theory. Given a first-order theory T, the minimal model M of T satisfies the following conditions:

- $\forall \phi \in T$, $M \models \phi$,
- $\forall M' \subset M$, $\exists \phi \in T$, $M \not\models \phi$.

The first condition states M satisfies all the sentences in T. The second condition states that there is no proper substructure M' of M that also satisfies all the sentences in T. We denote the minimal model of T as $MinModel(T)$.

3.1 Stage 1: Substate Property Generation

For one agent, each substate can only differ from either its belief base, its goal base, or both. Consequently, we define the substate property as the combination of the current beliefs and the desired goals of the agent. The current beliefs and the desired beliefs are defined as follows:

$$\begin{aligned} CB &::= I(B) \cup I(K), \\ DB &::= I(goals) \end{aligned}$$

CB is a first-order theory that derives current beliefs from its belief base B and knowledge base K, while DB denotes the desired beliefs. Following the fourth principle to interpreter *vGOAL* specifications, the interpretation of $goals$ is either the first goal base of the agent or empty. Substate properties involve both CB and DB through a predefined function F. F transforms the agent's desired beliefs into a new form that reflects those desired beliefs. Its formal definition is as follows:

$$F(G) ::= \begin{cases} \bigcup^{n} a\text{-}goal(g_i) & if\ G ::= g_1 ... g_n \text{ and } n > 0, \\ \emptyset & \text{otherwise.} \end{cases}$$

The substate properties are formally defined as follows:

$$subP ::= MinModel(CB \cup F(DB)).$$

3.2 Stage 2: Enabled Constraint Generation

The constraints that constrain an agent to generate feasible actions or sent messages are referred to as enabled constraints. Constrained by the current and desired beliefs, an agent generates decisions. The generated constraints are defined as follows:

$$\begin{aligned} EC &::= subP \cup I(C), \\ GC &::= MinModel(EC) \setminus subP. \end{aligned}$$

3.3 Stage 3: Enabled Action Generation

An action can be triggered only when a related enabled constraint and its preconditions are satisfied by the current beliefs. The generated actions are defined as follows:

$$\begin{aligned} EA &::= subP \cup GC \cup I(A), \\ GA &::= MinModel(EA) \setminus MinModel(subP \cup GC). \end{aligned}$$

3.4 Stage 4: Enabled Sent Message Generation

During a reasoning cycle, if the decision-making module fails to generate a feasible action, it will attempt to generate enabled sent messages for exchanging information with other agents. A message can be sent only when the related enabled constraint is satisfied. The enabled sent messages are defined as follows:

$$\begin{aligned} ES &:: = subP \cup GC \cup I(S), \\ GS &:: = MinModel(ES) \setminus MinModel(subP \cup GC). \end{aligned}$$

sub_info of the agent will be changed if GS is not an empty set. M_S will be assigned with GS, which is defined as follows:

$$M_S ::= GS.$$

3.5 Stage 5: Event Processing

In each reasoning cycle, each agent processes events including adopting subgoals to achieve the desired state, revising current beliefs, and responding to the received messages from the last reasoning cycle. The state of the multi-agent system may change as a result of the event processing altering the state of an agent. In the reasoning cycle, the received messages of an agent are denoted with M_R. The enabled event processing is defined as follows:

$$\begin{aligned} EP &:: = subP \cup M_R \cup I(P), \\ PR &:: = MinModel(EP) \setminus MinModel(subP \cup I(P)). \end{aligned}$$

If M_R is not an empty set, the sub_info of the agent will be altered. This modification occurs because M_R undergoes reinitialization, resetting it to an empty set after event processing, which is formally defined as follows:

$$M_R ::= \emptyset.$$

3.6 Stage 6: Communication

During each reasoning cycle, agents exchange information on the basis of the information of sub_info. To define the effects of communication of the sub_info of each agent, we utilize the following functions.

We utilize three functions to convert sent messages into their corresponding received messages. First, $inst(msg_s)$ instantiates the receivers of a sent message. Secondly, $Inst(S, M_S)$ instantiates all messages sent by an agent, using $inst(msg_s)$ as the basis. Third, $MP(S, msg_s)$ converts a sent message to its corresponding received message.

$$inst(msg_s) ::= \begin{cases} \bigcup^{r} I(msg_s)[R \mapsto r], \text{ if } R = all, \text{ and } r \in \bigcup id \\ \bigcup^{r} I(msg_s)[R \mapsto r], \text{ if } R = allother \text{ and } r \in \bigcup id \setminus S, \\ I(msg_s), \text{ if } R = id, \end{cases}$$

$$Inst(S, M_S) ::= \begin{cases} \emptyset, \text{ if } I(M_S) = \emptyset, \\ inst(msg_s \bigcup Inst(S, M_S) \setminus I(msg_s)), \text{ otherwise} \end{cases}$$

$$MP(S, msg_s) ::= \begin{cases} I(sent(S,p)), \text{ if } msg_s = send(r,p), \\ I(sent!(S,p)), \text{ if } msg_s = send!(r,p), \\ I(sent?(S,p)), \text{ if } msg_s = send?(r,p). \end{cases}$$

Next, we use three functions to update the subinfo of one agent. First, $P_1(sub\_info, S, msg_s)$ defines how an agent updates its sub_info for a single sent message. Second, $P_2(sub\_info, S, M)$ defines how an agent updates its sub_info for a set of sent messages, using $P_1(sub\_info, S, msg_s)$ as the basis. Third, $P_3(sub\_info)$ describes the initialization of M_S of an agent.

$$P_1(sub\_info, S, msg_s) ::= \begin{cases} sub\_info[M_R \mapsto M_R \bigcup MP(S, msg_s)], \text{ if id=r} \\ sub\_info, \text{ otherwise}, \end{cases}$$

$$P_2(sub\_info, S, M_S) ::= \begin{cases} sub\_info, \text{ if } I(M_S) = \emptyset \text{ or } id \neq r \\ P_2(P_1(sub\_info, S, msg_s), S, M_S \setminus msg_s), \text{ otherwise}, \end{cases}$$

$$P3(sub\_info) ::= sub\_info[M_S \mapsto \emptyset)].$$

We define the $state\_info$ as a collective set of the sub_info of each agent within the multi-agent system, denoting as $(sub\_info)_{\times n}$. Moreover, we use $(id : M_S)_{\times n}$ to denote the sent messages of each agent within the system during the current reasoning cycle. After the reasoning cycle of each agent, the update of $state\_info$ is formally defined as follows:

$$sub\_info \xrightarrow{(id:M_S)_{\times n}} (P_3((P_2(sub\_info, id, M_S))_{\times n}))_{\times n}.$$

3.7 State Update

For a multi-agent system, agents participate in a modular reasoning cycle and communicate with other agents during the final stage of the cycle. The state of the multi-agent system is updated once all agents have completed their current reasoning cycle. The substate of a multi-agent system, i.e., the state of an agent, can only be modified by generated actions, GA, and the processed results of the event processing, PR.

An agent changes its current belief base based on the rules of action effects and the enabled actions. The action effects will change the state of the agent, subsequently changing the state of the multi-agent system.

First, we define how an action changes the current belief base of the agent. The rules on action effects E are defined by $qrule_7$, which is in the form $hs_1 \rightarrow hs_2$. Both hs_1 and hs_2 follow the construction rule of hs in syntax, and $I(hs) ::= I(\bigwedge_m B_m \wedge \bigwedge_n \neg B_n)$. We define a function U to describe the belief updates incurred by actions as follows:

$$U(B, GA, E) :: = \begin{cases} I(B) \cup \bigcup^m I(\{B_m\}) \setminus \bigcup^n I(\{B_n)\}, \text{if} I(B) \cup GA \vdash I(hs_1), \\ I(B) \text{ ,otherwise.} \end{cases}$$

In each reasoning cycle, the agent can only generate either an enabled action or send messages, but it can handle all received messages. We define a function T to update *substate* based on action effects during each reasoning cycle, and T will not modify the substate if there is no enabled action effect.

For the generated action effect, the substate is updated as follows:

$$T(substate, GA) :: = \begin{cases} id : (U(B, GA, E), I(goals)), \text{ if } GA \neq \emptyset, \\ id : (I(B), I(goals)), \text{ if } GE = \emptyset, \end{cases}$$
$$substate :: = T(substate, GA).$$

A processed result of event processing can modify beliefs, goals, or both. Additionally, an instance of a *response* can take the form of either msg_s or *update*. It is worth noting that only an instance of *update* will modify the substate, which includes $insert(B, b)$, $delete(B, b)$, $adopt(goals, g)$, and $drop(goals, g)$.

For a processed result, the substate is updated as follows:

$$\begin{aligned} I(insert(B, b)) &:: = I(B) \cup b, \\ I(delete(B, b)) &:: = I(B) \setminus b, \\ I(adopt(goals, g)) &:: = I(goals) \cup g, \\ I(drop(goals, g)) &:: = I(goals) \setminus g, \\ H(S, r) &:: = \begin{cases} id : (I(insert(B, b)), I(goals)) & \text{if } r = insert(b), \\ id : (I(delete(B, b)), I(goals)) & \text{if } r = delete(b), \\ id : (I(B), I(adopt(goals, g))) & \text{if } r = adopt(g), \\ id : (I(B), I(drop(goals, g))) & \text{if } r = drop(g), \\ id : (I(B), I(goals)) & \text{, otherwise.} \end{cases} \end{aligned}$$

For the processed results of the event processing, PR, we define the function F to update the substate as follows:

$$F(S, PR) = \begin{cases} F((H(S, r), PR \setminus r) & \text{if } PR \setminus r \neq \emptyset, \\ S & otherwise. \end{cases}$$

Assuming a multi-agent system containing n agents ($n \geq 1$), the state is represented as $(substate)_{\times n}$. In each reasoning cycle, the substate can only be changed by the effects of enabled actions and the processed results of event processing. The effects of enabled actions corresponds to an action (Act), and the processed results of event processing involve the received messages of the current reasoning cycle (M_R). We use the $(id : (Act, M_R))_{\times n}$ to represent a transition that may change the substate, subsequently changing the state. The operational semantics of a *vGOAL* specification is defined as follows:

$$(substate)_{\times n} \xrightarrow{(id:(Act,M_R))_{\times n}} (F(T(substate, Act), PR)_{\times n},$$

where Act represents the generated action, and PR denotes the processed results of the event processing, involving processing the received messages M_R. Although GE and PR can both be empty for an agent, if any agent has a goal, at least one agent will have non-empty GE or PR. In our setting, if an agent fails to generate any decisions based on its current beliefs, it should send messages to other agents to obtain more information to accomplish its goal.

Moreover, if any *substate* is updated, each sub_info within the multi-agent system will be automatically adjusted, namely, the belief base and goal base will be modified to align with the *substate*.

4 Case Study

Using a real-world autonomous logistic system, we have validated our formal approach to safe autonomous decision-making. Accordingly, we use the system to explain how to use *vGOAL*.

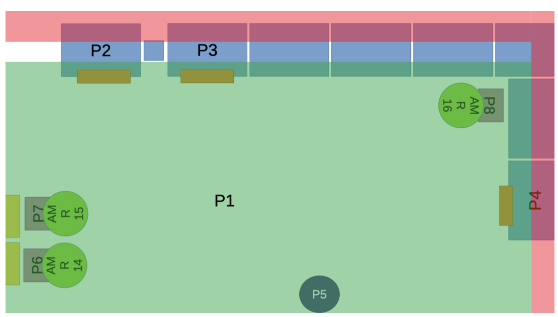

Fig. 1. Layout of the Robot Environment

The autonomous system is composed of three autonomous mobile robots, situated in the environment depicted in Figure 1. The case study aims to perform a collaborative transportation task. Non-red areas are considered safe places, denoting from P_1 to P_8, while red areas are considered unsafe, denoting by P_9. P_2 is the destination of the delivery task; P_3 and P_4 are the pick-up station; P_5 is a waiting point for the charging station; P_6, P_7, and P_8 are the charging stations; and P_1 is the other places except the aforementioned areas. The nine areas can be classified into four categories. Category I only contains P_1. The location of Category I is a safe place, but agents do not need permission to access it, and it has no dock. Category II includes P_2, P_3, P_4, P_6, P_7, and P_8. The locations of Category II are safe places, and agents need permission to access them. There is a dock for each location. Category III includes P_5. The location of Category III is a safe place, and agents need permission to access it, but it has no dock. Category IV only includes P_9. The location of Category IV is an unsafe place, and agents need to avoid moving there.

We demonstrate each key aspect of the *vGOAL* specifications using a subset of the specifications that specify the case study. For a comprehensive version of the formal specification for the case study, we refer readers to [18].

First of all, we have to determine how to specify agents within the multi-agent system. We need to define four agents in the case study: three for the real-world agents, designated as A1, A2, and A3, and one for a dummy agent, denoted as C. In our approach, we utilize a dummy agent to manage competing requests for critical resources, such as permissions for locations. The specification of the multi-agent system is specified as follows:

```
Agents = [A1,A2,A3,C],
```

where A1, A2, A3, and C are an instance of the agent class defined in the *vGOAL* interpreter.

To facilitate real-time autonomous decision-making, an agent will take both the real-time beliefs abstracted from sensor information and the prior beliefs as the complete belief base to make decisions. As it is common that not all required information can be sensed in practical scenarios, we need prior beliefs to specify the necessary but unperceived information, and it is shared by all agents within the system. The belief base of A3 and the prior beliefs of the system are specified as follows:

```
belief_base3  =[],
prior_beliefs=["on(1,3)","on(2,4)","on(3,3)","on(4,3)"].
```

Furthermore, the *vGOAL* interpreter receives real-time beliefs abstracted from sensor information on location, docking, and battery level. The initial complete belief base of A3 consists of the prior beliefs and the initial real-time beliefs, which is listed as follows:

```
belief_base3  =["on(1,3)","on(2,4)","on(3,3)","on(4,3)",
"at(8)",  "battery(2)","docked(8)","assigned(8)"].
```

An agent can have no goals, one goal, or multiple goals. Agent A3 has two goals, which are specified as follows:

```
goal_base3 = ['delivered(2,3)'],
goal_base4 = ["delivered(2,4)"],
goals3 = [goal_base3, goal_base4].
```

Dummy agents are used to manage critical resources. Their specifications are similar to those of real-world agents, including belief bases and goals. However, while real-world agents rely on sensor information to update their belief bases, dummy agents' belief bases are not affected by sensor information. Furthermore, dummy agents have no goals to pursue. The case study only requires one dummy agent, denoted as C, whose belief base and goals are listed as follows:

```
dummy_agents=["C"]
belief_base4 = ["idle(2)", "idle(3)", "idle(4)","idle(5)",
        "reserved(A1,6)", "reserved(A2,7)", "reserved(A3,8)"]
goals4 = []
```

The *vGOAL* interpreter provides a class for agents, whose attributes involve a unique identifier, a belief base, goals, sent messages, and received messages. The sent messages and received messages are empty by default. Therefore, users only need to specify an agent with the other three values. The specifications of Agent A3 and the dummy agent are specified as follows:

```
A3 = Agent("A3", belief_base3, goals3)
C = Agent("C", belief_base4, goals4)
```

A knowledge base is a collection of facts and rules that the decision-making module uses to reason about the world. In *vGOAL*, a knowledge base can contain either a first-order implication without negative recursion or a ground atom. Two representative rules in the knowledge base are specified as follows:

```
"forall w. on(w,4) implies available(w)",
"equal(charging,charging)".
```

vGOAL utilizes a set of rules, referred to as the constraints of action generation, to ensure that the generated decisions are moving towards a goal. These constraints are either related to the generation of actions or the generation of messages to acquire more information about the environment. Two representative constraints are specified as follows:

```
"forall w,y in D2 . a-goal holding(w) and docked(p) and not
holding(y) and docked(4) and available(w) implies A(w)",
"forall p,w in D2 . a-goal at(p) and not holding(w) and
                        not equal(p,2) implies S(p)".
```

The first constraint pertains to the action generation, and the second constraint pertains to the generation of sent messages. As mentioned in Section 2, users only need to specify the domain of variables that only occur on the left side of the implication due to the implementation of the interpreter.

In *vGOAL*, feasible actions are derived using a set of rules called the enabledness of actions, which requires including a generated constraint and may impose restrictions on the current belief base. Two of the enabledness of action generation are specified as follows:

```
"forall w. A(w) implies pickup(w)"
"forall p. exists y. C(p) and at(y) and equal(y,1) and
                  not equal(p,5) implies move1(y,p)".
```

The first rule only involves a generated constraint, whereas the second rule involves both a generated constraint and current beliefs.

In *vGOAL*, sent messages are derived using a set of rules, which only includes a generated constraint and may impose restrictions on the current belief base. One rule for the generation of sent messages is specified as follows:

```
"forall p. S(p) implies send!(C) idle(p)".
```

vGOAL includes rules related to event processing, which encompasses responding to received messages and adopting subgoals of the first goal base on the basis of current beliefs. Five rules for event processing are specified as follows:

```
"fatal implies drop all",
"forall z. exists x,y. sent!(x) at(y) and reserved(x,z)
            and not equal(z,y) implies insert idle(z)",
"forall x. exists y. sent!(x) idle(y) and reserved(x,y)
                        implies send:(x) assigned(y)",
"exists x,y. sent!(x) idle(y) and reserved(z,y) and
                   equal(x,z) implies delete idle(y)",
"exists x,w,p. a-goal on(w,2) and on(w,p) and at(x)
                                  implies adopt at(p)".
```

The first rule states that all goals should be dropped if a fatal error occurs. The next three rules illustrate three distinct approaches to responding to a received message, including belief insertion, message sending, and belief deletion. The last rule specified how to adopt a subgoal toward the desired goal.

vGOAL employs action effects to determine how to modify the current belief base. These effects can either involve belief insertion or deletion. As a result, the associated rule may involve negative recursion, a property not shared by rules in other components. An example rule for the generation of action effects is provided below:

```
"pickup": "forall w,p,y in D2 . pickup(w) and not holding(y)
           and on(w,p) implies holding(w) and not on(w,p)"
```

Moreover, the real-time information can include error messages, necessitating error handling. We emphasize that our framework can conveniently handle errors. In another word, users can simply specify how to handle errors in the specifications without changing any implementation of the framework. In the case study, we identify four types of errors: E_1, *dock* errors; E_2, *pick up* errors; E_3, $drop\ off$ errors; and E_4, *charge* errors. In our setting, the non-fatal errors are E_1, E_2, and E_3, and the fatal errors are E_4, which is specified in the knowledge base as follows:

```
"E1 implies nonfatal",
"E2 implies nonfatal",
"E3 implies nonfatal",
"E4 implies fatal",
```

If an agent encounters a fatal error, it should send a message to the dummy agent to report its current location. If an agent encounters a nonfatal error, we need a dummy rule to avoid any meaningful constraints. Therefore, two constraints on error handling are specified as follows:

```
"forall p. at(p) and fatal implies M(p)",
"nonfatal implies Dummy",
```

If an agent encounters a fatal error, the agent will be considered broken and will drop all goals and beliefs. If an agent encounters a non-fatal error, it will drop the focused goals and adopt new goals. After inserting new goals, it will delete corresponding nonfatal errors to enter the next reasoning cycle. The rules on error handling are specified in the event processing as follows:

```
"fatal implies drop all",
"fatal implies delete all",
"nonfatal and not goal_change implies drop all",
"nonfatal and not goal_change implies adopt located(charging)",
"nonfatal and not goal_change implies adopt at(5)",
"nonfatal and not goal_change implies insert goal_change",
"nonfatal and E1 implies delete E1",
"nonfatal and E2 implies delete E2",
"nonfatal and E3 implies delete E3",
```

5 Discussion

The motivation of *vGOAL* is the generation of verifiably safe decision-making for autonomous systems. Consequently, it is pertinent to conduct a comparison with the APLs capable of generating verified decisions. In this section, we discuss the key aspects of *vGOAL*, along with a comparison with GOAL, Gwendolen, and AgentSpeak (Jason).

vGOAL stands out from GOAL, Gwendolen, and AgentSpeak (Jason) in generating safe decisions without the need for additional computation. As discussed in Section 3.1, the first stage of each reasoning cycle involves generating the substate property, which links each state to a state property. Hence, we can prove that a state satisfies its safety properties by showing that all safety properties are contained within the state properties without additional computation. However, GOAL and AgentSpeak necessitate formal specifications of the original programming language and verification tools [1] [13], while Gwendolen relies on the Agent Java PathFinder (AJPF) for model checking, thereby encountering efficiency problems [9].

Durative action modeling and error handling are crucial and challenging issues in autonomous decision-making. Notably, we address the challenge of error detection in a different way than GOAL, Gwendolen, and Jason. Specifically, *vGOAL* logically handles errors by separating error detection from the decision-making module and allowing users to specify how to handle errors in the specifications without modifying the implementation of the framework. In contrast, error handling is hard-coded into the implementation of Gwendolen and Jason, requiring users to modify the implementation to specify how to handle action failures [2] [16]. While GOAL does not have a specific error-handling mechanism, it can recognize action failure by comparing received perceptions with desired effects. In practice, the method involves comparing the received perceptions with the desired effects [12] [14], which can be laborious to identify all potential situations of action failure.

Despite being based on speech-act theory, the communications of all four languages have different performatives. *vGOAL* and GOAL employ the least performatives, namely indicative, declarative, and interrogative, which do not directly alter current goals [11]. In contrast, Gwendolen utilizes performatives such as *tell*, *perform*, and *achieve*, which directly affect intentions [7]. Jason employs more performatives, compared with *vGOAL*, GOAL, and Gwendolen [2]. In summary, *vGOAL* and GOAL use a simpler communication mechanism than Gwendolen and Jason, employing mailbox semantics without direct modification of goals. Notably, in *vGOAL*, the communication component is encoded in a first-order logical manner to allow automated minimal model generation.

The implementation of the interpreter for *vGOAL* is in Python, which differs from the implementation of the interpreters for GOAL, Gwendolen, and AgentSpeak in Java. *vGOAL* has the advantage that only it can be readily encoded in a decision-making node in ROS, compared with GOAL, Gwendolen, and AgentSpeak. *vGOAL* has already been integrated with ROS using rosbridge, as well as Gwendolen and AgentSpeak [5]. Additionally, there is currently no known research that connects GOAL with ROS.

6 Conclusion

To achieve verifiably safe autonomous decision-making, we have developed an innovative formal approach based on *vGOAL*. In this paper, we aim to give a comprehensive introduction to *vGOAL*, as it is pivotal in presenting and justifying our formal approach to safe autonomous decision-making. Initially, we presented its formal syntax and operational semantics, providing a solid foundation for formal verification. To demonstrate the applicability of the language, we described a real-world autonomous logistic system that has been validated using *vGOAL* and its interpreter. Finally, we compared the key aspects of *vGOAL* with comparable APLs to demonstrate its advantages. In the future, we aim to enrich the case studies of *vGOAL* with numerous complicated real-world autonomous systems. Moreover, we intend to conduct an empirical analysis to compare *vGOAL* with GOAL, Gwendolen, and AgentSpeak (Jason). We believe *vGOAL* can be highly valuable for developing safe autonomous robotic applications.

Acknowledgements

This research is partially funded by the Research Fund KU Leuven. We thank Jens Vankeirsbilck for providing Fig.1.

References

1. Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking AgentSpeak. In: Proceedings of the second international joint conference on Autonomous agents and multiagent systems. pp. 409–416 (2003)
2. Bordini, R.H., Hübner, J.F.: BDI agent programming in AgentSpeak using Jason. In: International workshop on computational logic in multi-agent systems. pp. 143–164. Springer (2005)
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. John Wiley & Sons (2007)
4. Cardoso, R.C., Ferrando, A.: A review of agent-based programming for multi-agent systems. Computers **10**(2), 16 (2021)
5. Cardoso, R.C., Ferrando, A., Dennis, L.A., Fisher, M.: An interface for programming verifiable autonomous agents in ROS. In: Multi-Agent Systems and Agreement Technologies, pp. 191–205. Springer (2020)
6. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is coming: A modern probabilistic model checker. In: International Conference on Computer Aided Verification. pp. 592–600. Springer (2017)
7. Dennis, L.A.: Gwendolen semantics: 2017 (2017)
8. Dennis, L.A., Farwer, B.: Gwendolen: A BDI language for verifiable agents. In: Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning, Society for the Study of Artificial Intelligence and Simulation of Behaviour. pp. 16–23. Citeseer (2008)
9. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Automated software engineering **19**(1), 5–63 (2012)
10. Hindriks, K.V.: Programming rational agents in GOAL. In: Multi-agent programming, pp. 119–157. Springer (2009)
11. Hindriks, K.V.: Programming Cognitive Agents in GOAL. Vrije Universiteit Amsterdam (June 2021)
12. Hindriks, K.V., Dix, J.: GOAL: a multi-agent programming language applied to an exploration game. Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks pp. 235–258 (2014)
13. Jensen, A.B., Hindriks, K.V., Villadsen, J.: On using theorem proving for cognitive agent-oriented programming. In: 13th International Conference on Agents and Artificial Intelligence. pp. 446–453. Science and Technology Publishing (2021)
14. Jensen, A.B., Villadsen, J.: GOAL-DTU: development of distributed intelligence for the multi-agent programming contest. In: The Multi-Agent Programming Contest 2019: Agents Assemble–Block by Block to Victory 14. pp. 79–105. Springer (2020)
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. pp. 200–204. Springer (2002)
16. Stringer, P., Cardoso, R.C., Dixon, C., Dennis, L.A.: Implementing durative actions with failure detection in Gwendolen. In: Engineering Multi-Agent Systems: 9th International Workshop, EMAS 2021, Virtual Event, May 3–4, 2021, Revised Selected Papers. pp. 332–351. Springer (2022)
17. Yang, Y.: Verifiably safe decision-making for autonomous systems. In: Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems. pp. 2973–2975 (2023)

18. Yang, Y.: vGOAL. https://kuleuven-my.sharepoint.com/:f:/g/personal/yi_yang_kuleuven_be/EjUTI-DUvkdBlBKoNWxcVgIB8GMfhyAZHSA_i1b7ovskqw?e=k6FINj (2023)
19. Yang, Y., Holvoet, T.: Generating safe autonomous decision-making in ROS. In: Fourth Workshop on Formal Methods for Autonomous Systems. vol. 371, pp. 184–192. Open Publishing Association (9 2022)
20. Yang, Y., Holvoet, T.: Making model checking feasible for GOAL. In: 10th International Workshop on Engineering Multi-Agent Systems (2022)