Enhancing Confidence of the *vGOAL* Interpreter Using SAT Solving

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. Agent programming languages and their interpreters are crucial in autonomous decision-making. While formal methods are extensively utilized to ensure the correctness of agent programs, their application for verifying the implementation correctness of interpreters remains infrequent. To formally specify and verify autonomous decision-making, we proposed *vGOAL* and implemented its interpreter. The implementation correctness of the *vGOAL* interpreter is crucial for users to gain trust in the *vGOAL* approach. Using program verification is one option, yet this would require a huge effort to verify the correctness of the *vGOAL* interpreter.
In this paper, we propose integrating an SAT-solving component into the *vGOAL* interpreter to enhance confidence in its core component: minimal model generation. The SAT-solving component consists of two subcomponents: an SAT-encoding component and an SAT solver. Leveraging PySAT for its interface to advanced solvers, our main contribution lies in the SAT encoding. We devise an algorithm to encode the inputs and outputs of the core component into a satisfiable CNF formula. Importantly, we justify that this algorithm generates a satisfiable CNF formula only if the result is correct. We demonstrate the practicality and efficiency of this SAT-solving approach using a case study involving an autonomous transportation system with three mobile robots.

Keywords: Autonomous Decision-Making · Implementation Correctness · *vGOAL* interpreter · SAT Solving

1 Introduction

Autonomous systems, defined as entities capable of independent task completion without continuous human instructions, hold immense potential for saving lives and mitigating risks in hazardous environments like nuclear power plants and space exploration. It is important to convince the public that autonomous systems will correctly execute tasks as desired without any violations of safety requirements. The development of safe autonomous decision-making is a challenging task in developing autonomous systems.

For many years, agent programming languages (APLs) and their interpreters have been extensively researched to develop autonomous decision-making [12]. More precisely, an APL is a specification language to specify an autonomous

decision-making mechanism, and an APL interpreter serves as an autonomous decision-making component. Therefore, we can increase confidence in the autonomous decision-making process using APLs from two aspects: APL programs and APL interpreters.

Formal verification is built on a rigorous mathematical foundation. Hence, it is a reliable tool to provide high confidence for autonomous systems, especially in safety-critical applications. Many efforts have been made to prove the correctness of an APL program using formal verification, enhancing the confidence of the APL program. Particularly, model checking is the most successful and influential verification method in verifying APLs, including AgentSpeak [2], Gwendolen [4], and GOAL [7], owing to the automated verification process [1], [5], [9], [13]. On the other hand, there is a noticeable gap in verifying the implementation correctness of APL interpreters. To the best of our knowledge, the implementation correctness of APL interpreters has only been briefly discussed in [5], and the authors believe it is important but requires significant effort.

In our previous work, we proposed *vGOAL* [17], a specification language specifically designed to formally specify and verify safe autonomous decision-making, and implemented its interpreter [16]. To verify the correctness of *vGOAL* specifications, we have implemented an automated model-checking process for *vGOAL*, and its preliminary version is described in [15]. The *vGOAL* interpreter is a tool to generate autonomous decisions. Consequently, its implementation correctness is crucial for users to trust the *vGOAL* approach. This paper aims to enhance the confidence of the *vGOAL* interpreter using an efficient and automated formal method.

It is truly a significant task to specify all preconditions and postconditions of thousands of code lines to verify the implementation correctness of the *vGOAL* interpreter. Instead of verifying the generation process, we propose demonstrating the correctness of the generated outputs, thereby enhancing confidence in the *vGOAL* interpreter. To achieve this, we turn to Boolean Satisfiability Problem (SAT) encoding and solving. SAT is the problem of determining if there exists an interpretation that satisfies a given Boolean formula [10]. SAT solvers have been researched and developed for many years, there are many well-known efficient SAT solvers such as Chaff [11]. If software verification problems are converted into Boolean Satisfiability Problems, the SAT Solvers can make the automated execution of software verification possible [6]. Therefore, SAT solving is a feasible solution to increase the confidence of an APL interpreter while adhering to the efficiency and automation requirements.

The *vGOAL* interpreter implements the semantics of *vGOAL*, whose core component is the minimal model generation. Consequently, the implementation correctness of the minimal model generation component plays a crucial role in the implementation correctness of the *vGOAL* interpreter. One distinguishing feature of *vGOAL* is its direct conversion of specifications into equivalently expressive propositional logical specifications. This characteristic facilitates the integration of an SAT-solving component into the *vGOAL* interpreter, particularly into its core component, the minimal model generation.

To practically enhance the confidence of the *vGOAL* interpreter, we developed and integrated an SAT-solving component into its core component: minimal model generation. The SAT-solving component encompasses two subcomponents: an SAT-encoding component and an SAT solver, leveraging PySAT [8] as a simple interface to numerous state-of-the-art SAT solvers.

Our main contribution lies in SAT encoding. Specifically, we devise an algorithm to encode the inputs and outputs of the core component into a satisfiable conjunctive normal form (CNF) formula. Moreover, we justify that this algorithm generates a satisfiable CNF formula only if the result is correct. Each output generated by the minimal model generation process is checked by the SAT-solving component. In the event of any inconsistencies detected during this process, the *vGOAL* interpreter halts execution. The SAT-solving component enhances the confidence of the *vGOAL* interpreter because no incorrectly inferred autonomous decision will be generated. We demonstrate the practicability and efficiency of the SAT-solving component through an autonomous transportation system including three autonomous mobile robots.

The rest of the paper is structured as follows. Section 2 briefly introduces *vGOAL*, including its definitions and reasoning cycle. Section 3 describes how to integrate the SAT-solving component into the *vGOAL* interpreter. Section 4 presents the formulation of the encoding problem, the description of an encoding algorithm, and the justification of the SAT-encoding algorithm. Section 5 describes a case study where we empirically test the differences of using the *vGOAL* interpreter with or without the SAT-solving component. Finally, we draw conclusions on our work.

2 Preliminaries

This section aims to explain why the minimal model generation component is the core component of the *vGOAL* interpreter. We clarify all key concepts in the paper, and we refer interested readers to [17] for more details of *vGOAL*.

2.1 *vGOAL*

Definition 1. *(vGOAL Specifications) [17] A* vGOAL *specification is defined as*

$$vGOALSpec ::= (MAS, K, C, A, S, P, E, D)$$
$$MAS ::= (id, B, goals, M_S, M_R)^*$$

A *vGOAL* specification specifies autonomous decision-making. The first main component is agents' specifications, MAS. Each agent's specification comprises a unique identifier, a belief base, a set of goals, sent messages, and received messages, denoted as id, B, $goals$, M_S, and M_R, respectively. The other specifications are system specifications. K represents the knowledge base; C denotes the rules on enabled constraints generation; A denotes the rules on feasible action generation; S denotes the rules on sent message generation; P denotes the rules

on event processing, including modifying agent goals and beliefs, and processing received messages; E denotes action effects; and D represents the domain of all variables.

Table 1. Semantics of *vGOAL* Specifications

Specification	Syntax	Semantics
B	$[b_1, ..., b_n]$	ground atoms: $I(B) = \{b_1, ..., b_n\}$
$goals$	$[[g_{11}, ..., g_{1\mathrm{m}}], ..., [g_n]]$	ground atoms: $I(goals) = \{a\text{-}goal\text{-}g_{11}, ..., a\text{-}goal\text{-}g_{1\mathrm{m}}\}$
M_S	$[s_1, ..., s_n]$	atoms: $I(M_S) = \{s_1, ..., s_n\}$
M_R	$[r_1, ..., r_n]$	atoms: $I(M_R) = \{r_1, ..., r_n\}$
K	$[k_1, ..., k_n]$	a first-order theory: $I(K) = \{k_1, ..., k_n\}$
C	$[c_1, ..., c_n]$	a first-order theory: $I(C) = \{c_1, ..., c_n\}$
A	$[a_1, ..., a_n]$	a first-order theory: $I(A) = \{a_1, ..., a_n\}$
S	$[s_1, ..., s_n]$	a first-order theory:$I(S) = \{s_1, ..., s_n\}$
P	$[p_1, ..., p_n]$	a first-order theory: $I(P) = \{p_1, ..., p_n\}$
id	id	$I(id) = id$
D	D	$I(D) = D$
E	E	$I(id) = E$

Table 1 illustrates the semantics of *vGOAL* specifications, detailing their syntax and corresponding interpretations: B and $goals$ are interpreted as sets of ground atoms; M_S and M_R are interpreted sets of atoms; K, C, A, S, and P are interpreted as first-order theories; id, D, and E maintain their identity as specified. This table demonstrates that all *vGOAL* specifications, except for id, D, and E, are expressed within first-order logic. Additionally, D plays a crucial role in the transformation of first-order logical expressions to their logically equivalent logical formulae by removing all variables.

Definition 2. *(*vGOAL *States) [17] A* vGOAL *state is formalized as follows:*

$$state ::= (substate)_{\times n},$$
$$substate ::= id{:}(I(B), I(goals)).$$

The *vGOAL state* of a system is formally defined as a composition of substates, $(id{:}(I(B), I(goals)))_{\times n}$. Each substate represents an agent with a unique identifier and the semantics of its beliefs and goals, denoted as $I(B)$ and $I(goals)$.

Definition 3. *(Operational Semantics of* vGOAL*) [17]*

$$(substate)_{\times n} \xrightarrow{(id:(M_R, Act))_{\times n}} (substate')_{\times n}.$$

The operational semantics of *vGOAL* is established by the reasoning cycle, which involves the minimal model generation of first-order theories and function updates based on the interpretation of *vGOAL* specifications. After each reasoning cycle, a substate can only be updated by the action effects and the processed

results of the event processing, including processing the received messages, subsequently updating the *vGOAL* state.

2.2 *vGOAL* Interpreter

vGOAL is a specification language for autonomous decision-making mechanisms, and it requires an interpreter to generate decisions based on the given *vGOAL* specification. The *vGOAL* interpreter implements the operational semantics of *vGOAL*, serving as an agent-based decision-making component for autonomous systems [16].

Table 2. State Update in *vGOAL* Interpreter

Stage	Input	Output	Process
1	MAS	$state$	Interpretation
2.0	$id : (B, goals)$	$substate$	Start substate update
2.1.a	K, B, $goals$, D	K',B, $goals$	Logical Equivalence Transformation
2.1.b	K', B, $goals$	$subP$	Minimal Model Generation
2.2.a	$subP$, C, D	$subP$, C'	Logical Equivalence Transformation
2.2.b	$subP$, C'	GC	Minimal Model Generation
2.3.a	$subP$, GC, A, D	$subP$, GC, A'	Logical Equivalence Transformation
2.3.b	$subP$,GC,A'	GA	Minimal Model Generation
2.4.a	$subP$, GC, S, D	$subP$, GC, S'	Logical Equivalence Transformation
2.4.b	$subP$, GC, S'	GS	Minimal Model Generation
2.5.a	$subP$, M_R, P, D	$subP$, M_R, P'	Logical Equivalence Transformation
2.5.b	$subP$, M_R, P'	PR	Minimal Model Generation
2.6	MAS	MAS'	Communication
2.7	$substate$, PR, E, D	$substate'$	Substate Update
3	$(substate')_{\times n}$	$state'$	Substate Combination

Table 2 presents an overview of the state update implemented in the *vGOAL* interpreter. As an agent-based autonomous system consists of agents in a modular manner, a state consists of substates in a modular manner. The state update involves three stages. At first, *state*, the current state, is directly interpreted by MAS, agent specifications. Each substate of *state* needs to go through the second stage: substate update. After each substate of the *state* goes through the second stage, the state will be updated by the combination of the updated substates.

As presented in Table 2, the second stage is the key implementation of the *vGOAL* interpreter. The reasoning cycle involves six stages. The first five stages are implemented by the logical equivalence transformation and minimal model generation, and the last stage is implemented by communication among agents. The logical equivalence transformation is used to transform a given first-order theory under *vGOAL* syntax to its logically equivalent first-order theory without variables, specifically, from K and D to K', from C and D to C', from

A and D to A', from S and D to S', and from P and D to P'. Hence, the converted first-order theories are expressible in propositional logic. The minimal model generation is used to generate the substate properties($subP$), the generated constraints (GC), the generated actions (GA), the generated sent messages (GS), and the processed results (PR). The communication is implemented by exchanging messages among agents. Compared with the implementation of logical equivalence transformation and communication, the implementation of the minimal model generation is much more complex and error-prone. The confidence of the *vGOAL* interpreter will be significantly enhanced if the implementation correctness of the minimal model generation is guaranteed.

3 SAT Solving Integration

This section elaborates on the data flow and the workflow concerning the integration of SAT solving into the *vGOAL* interpreter. As explained in Section 2.2, the *vGOAL* interpreter implements the operational semantics of *vGOAL*, particularly through the implementation of the reasoning cycle that enables substate updates. The SAT-solving component is used to check the correctness of the generated results of the minimal model generation, thereby enhancing the overall confidence in the *vGOAL* interpreter. The implementation of the *vGOAL* interpreter embedded with the SAT solver is available at [14].

Table 3. Dataflow between the Reasoning Cycle and the SAT-Solving Component

Component	Precondition	Input	Output	Guarantee	Termination
2.1.a	No	K, B, $goals$, D	K',B, $goals$	\	No
2.1.b	No	K', B, $goals$	R_1	$R_1 = subP$?	No
SAT Solving	No	K', B, G,R_1	SAT	$R_1 = subP$	No
SAT Solving	No	K', B, G,R_1	$UNSAT$	$R_1 \neq subP$	Yes
2.2.a	SAT	$subP$, C, D	$subP$, C'	\	No
2.2.b	No	$subP$, C'	R_2	$R_2 = GC$?	No
SAT Solving	No	$subP$, C', R_2	SAT	$R_2 = GC$	No
SAT Solving	No	$subP$, C', R_2	$UNSAT$	$R_2 \neq GC$	Yes
2.3.a	SAT	$subP$, GC, A, D	$subP$, GC, A'	\	No
Stage 2.3.b	No	$subP$,GC,A'	R_3	$R_3 = GA$?	No
SAT Solving	No	$subP$,GC,A', R_3	SAT	$R_3 = GA$	No
SAT Solving	No	$subP$,GC,A', R_3	$UNSAT$	$R_3 \neq GA$	Yes
2.4.a	SAT	$subP$, GC, S, D	$subP$, GC, S'	\	No
2.4.b	No	$subP$, GC, S'	R_4	$R_4 = GS$?	No
SAT Solving	No	$subP$, GC, S', R_4	SAT	$R_4 = GS$	No
SAT Solving	No	$subP$, GC, S', R_4	$UNSAT$	$R_4 \neq GS$	Yes
2.5.a	SAT	$subP$, M_R, P, D	$subP$, M_R, P'	\	No
2.5.b	No	$subP$, M_R, P'	R_5	$R_5 = PR$?	No
SAT Solving	No	$subP$, M_R, P',R_5	SAT	$R_5 = PR$	No
SAT Solving	No	$subP$, M_R, P',R_5	$UNSAT$	$R_5 \neq PR$	Yes
2.6	SAT	MAS	MAS'	\	No

Table 3 outlines the data flow between the reasoning cycle of the *vGOAL* interpreter and the SAT-solving component. The table provides information on components, along with their preconditions, inputs, outputs, guarantees, and termination. Components include all stages in the reasoning cycle that interact with the SAT-solving component, involving from Stage 2.1 to Stage 2.6 and the SAT-solving component. Preconditions are the preconditions to enter the component. Inputs and outputs are the inputs and outputs of the component. Guarantees are the guarantees provided by the SAT-solving component. Termination gives information on whether the reasoning process should immediately terminate due to the erroneous results generated by the minimal model generation.

For the details of the inputs and outputs of the stages in the *vGOAL* interpreter, we explained in Section 2.2. The SAT-solving component takes the inputs and outputs of its previous stage as the inputs, and it generates the satisfiability result of inputs: either SAT or $UNSAT$.

The SAT-solving component is used to check the correctness of the results generated by the minimal model generation for the given input, involving stages 2.1.b, 2.2.b, 2.3.b, 2.4.b, and 2.5.b. If the SAT-solving component generates $UNSAT$, the *vGOAL* interpreter will immediately terminate. The preventive termination guarantees that the *vGOAL* never executes any erroneous decisions due to the erroneous minimal model generation. Only when the SAT-solving component generates SAT, the *vGOAL* interpreter proceeds to its subsequent reasoning stage, including 2.2.a, 2.3.a, 2.4.a, 2.5.a, and 2.6.

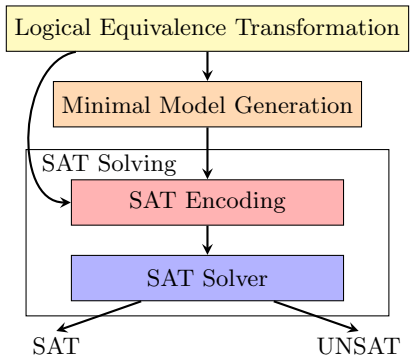

Fig. 1. Integration of the SAT Solving Component into the *vGOAL* Interpreter

Figure 1 illustrates the workflow of integrating SAT solving into the *vGOAL* interpreter. As shown in Table 1, the first five reasoning stages involve logical equivalence transformation and minimal model generation, where the SAT-solving component is integrated. The SAT-solving component comprises two subcomponents: an SAT-encoding component and an SAT solver. The inputs of the SAT encoding component are both inputs and outputs of the minimal model generation. The SAT-encoding component transforms these inputs into

their logically equivalent formulas in CNF. The SAT solver takes the generated CNF formula as input and outputs its satisfiability result. The output SAT indicates the satisfiability of the inputs, verifying the correctness of the results of the minimal model generation for the given inputs. Conversely, an output of $UNSAT$ denotes unsatisfiability, suggesting issues with the minimal model generation process in achieving the minimal model for the given inputs. We use PySAT [8] as the SAT solver for its simple access to numerous state-of-the-art SAT solvers. Consequently, our main work on the SAT-solving integration is the SAT-encoding component.

4 SAT Encoding

This section explains the SAT-encoding component in detail. First, we formula the encoding problem. Second, we describe the SAT-encoding algorithm. Finally, we justify the SAT encoding algorithm.

Definition 4 formally defines the inputs of the SAT-encoding component. The inputs of the SAT-encoding component are the inputs and outputs of the minimal model generation. T denotes the inputs of the minimal model generation, and M denotes the outputs of the minimal model generation.

Definition 4. *(Inputs of SAT-Encoding Component)*

$$
\begin{aligned}
atom &::= ground\_atom \\
atoms &::= \{atom\} \cup atoms | \emptyset \\
lhs &::= atom | atom \wedge lhs | \neg atom \wedge lhs \\
rule &::= lhs \rightarrow atom \\
rules &::= \{rule\} \cup rules | \emptyset \\
T &::= rules \cup atoms \\
M &::= M \cup \{atom\} | \emptyset \\
Input &::= T \cup M
\end{aligned}
$$

The inputs, T, are the first-order theory that is constrained by the $vGOAL$ syntax [17] with no variables and no quantification. The inputs of the $vGOAL$ interpreter is a $vGOAL$ specification that follows the $vGOAL$ syntax. The logical equivalence transformation converts a first-order theory to its logically equivalent first-order theory by removing all qualifications and variables. The outputs, M, are the consequence of the minimal model generation during the reasoning cycle. Therefore, it only contains ground atoms.

Definition 5 formally defines the output of the SAT-encoding component as a formula in CNF. A CNF formula is commonly represented in a set of sets. For example, a CNF formula, $(a_1 \vee a_2) \wedge b_1 \wedge (c_1 \vee c_2)$ is represented as $\{\{a_1, a_2\}, \{b_1\}, \{c_1, c_2\}\}$. The input of PySAT uses the common representation of a CNF formula.

Definition 5. *(Output of the SAT-Encoding Component)*

$$\begin{aligned} atom &::= ground\_atom | \neg ground\_atom \\ Disjunction &::= atom | atom \vee Disjunction \\ CNF &::= Disjunction \wedge CNF | Disjunction \\ clause &::= \{atom\} \cup clause | \emptyset \\ output &::= \{cluase\} \cup output | \emptyset \end{aligned}$$

Our goal is to prove the model generated by the minimal model generation process is the minimal model for the given *vGOAL* specification. Specifically, we need to encode the inputs of the SAT-encoding component into a formula in CNF that encodes the condition of the minimal model. As CNF formulae can be directly processed by an SAT solver, we can take advantage of the existing efficient SAT solvers. To achieve the goal, we design and implement Algorithm 1, which is the core of the SAT-encoding component.

Algorithm 1 takes the input of the SAT-encoding component as the input, denoting *input*. The algorithm generates a formula in CNF as the output, denoting CNF. Following Definition 4, *input* consists of a first-order theory (T) and a set of atoms (M). Following Definition 5, CNF is a CNF formula represented as a set of sets.

Lines 1-10 describe the initialization of G, A, $Check$, CNF, and Min_G. G denotes the set of the generated atoms by the minimal model generation process, initialized with $M \setminus atoms$. A denotes a set of negative atoms, initialized with an empty set. $Check$ denotes a set of the generated atoms that need to be checked if they are generated by at least one rule in T, initialized with G. CNF denotes a formula in CNF. Each atom in $atoms \cup M$ is included in CNF as a clause. Min_G denotes a dictionary that contains all possible conditions for the derivation of the generated atoms, initialized with an empty dictionary. The keys of Min_G are assigned with the generated atoms. Each value of the key in Min_G records all possible combinations of premises of the generated atom, initialized with an empty set.

Lines 11-31 extend CNF by encoding each rule in T into two conditions. *derived_atom* denotes the atom that can be generated by the given rule. $flag$ denotes if *derived_atom* is an atom in the set of generated atoms, G. If *derived_atom* is an atom of G, $flag$ is assigned with $True$. In this case, *derived_atom* is removed from the set $Check$, as there is at least one rule that can generate *derived_atom*. $Premise$ is a set of atoms that includes all preconditions to generate *derived_atom*.

Lines 19-20 describe the first condition is the transformation from the given rule to its logically equivalent formula in CNF. Following Definition 4, a rule in T can be transformed to logically equivalent clauses of a formula in CNF as follows:

$$\bigwedge_{i=1}^{n} A_i \rightarrow atom \equiv \bigwedge_{i=1}^{n} (\neg A_i \vee atom),$$

where $A_i ::= atom | \neg atom$.

Algorithm 1: Encode the input of the SAT-solving component into a formula in CNF

Input: $input = T \cup M$, $T = rules \cup atoms$
Output: CNF

$G \leftarrow M \setminus atoms$, $A \leftarrow \emptyset$
$Check \leftarrow G$
$CNF \leftarrow \emptyset$
for *each* $atom \in atoms \cup M$ **do**
    $CNF \leftarrow CNF \cup \{\{atom\}\}$

$Min\_G \leftarrow \emptyset$
for *each* $atom \in M$ **do**
    if $atom \in G$ **then**
        $Min\_G \leftarrow Min\_G \cup \{atom : \emptyset\}$
for *each* $rule ::= lhs \rightarrow atom \in rules$ **do**
    $derived\_atom \leftarrow atom$
    $flag \leftarrow False$
    if $derived\_atom \in G$ **then**
        $flag \leftarrow True$
        if $derived\_atom \in Check$ **then**
            $Check \leftarrow Check \setminus derived\_atom$
    $Premise \leftarrow \emptyset$
    for *each* $a ::= atom | \neg atom \in rule$ **do**
        $clause \leftarrow \{-a, derived\_atom\}$
        $CNF \leftarrow CNF \cup \{clause\}$
        if $atom \notin M \cup atoms$ **then**
            $A \leftarrow A \cup \{atom\}$
        $Premise \leftarrow Presmise \cup \{a\}$
    if *flag* **then**
        if $Min\_G[derived\_atom] = \emptyset$ **then**
            $Min\_G[derived\_atom] \leftarrow \{Premise\}$
        else
            $Min\_G[derived\_atom] \leftarrow Min\_G[derived\_atom] \times Premise$
for *each* $atom \in Min\_G$ **do**
    $CNF \leftarrow CNF \cup Min\_G[atom]$
if $Check \neq \emptyset$ **then**
    for *some* $atom \in Check$ **do**
        $CNF \leftarrow \{\{atom\}, \{\neg atom\}\}$
else
    for *each* $atom \in A$ **do**
        $CNF \leftarrow \{\{\neg atom\}\} \cup CNF$
return CNF

Lines 24-31 encode the second condition that every generated atom can be generated by at least one rule in T, which is the key to proving the generated model is the minimal model. First, all preconditions of the rule are added to $Premise$. If $flag$ is $True$, the Min_G is extended by two cases. If Min_G is an empty set, Min_G is assigned with $Premise$. If Min_G is not empty, Min_G is extended by all possibility that the original clauses multiply $Premise$. CNF will be extended by all clauses recorded in Min_G.

Lines 32-34 describe the case that at least one generated atom cannot be derived by T. In this case, the generated model is certainly not a minimal model of T. CNF is defined by two unsatisfiable clauses, $\{atom\}$ and $\{\neg atom\}$.

Lines 35-37 describe the extension of CNF. For each atom in A, its negative atom is added to CNF.

Following Definition 4, there are three kinds of relations between T and M: M is the minimal model of T; M is a model of T, but M is not the minimal model of T; M is not model of T. We illustrate how Algorithm 1 handles these three cases with three simple examples.

Example 1. Input: $T \cup M$, where $T = rules \cup atoms$,
$rules = \{rule_1, rule_2, rule_3\}$, $atoms = \{a_1, a_2\}$, $M = \{a_1, a_2, e\}$
$rule_1 = a_1 \wedge a_2 \rightarrow e$, $rule_2 = b_1 \wedge b_2 \rightarrow e$, $rule_3 = c \wedge \neg d \rightarrow f$
Step 1: Initialization
$G \leftarrow \{e\}$, $A \leftarrow \emptyset$, $Check \leftarrow G$, $CNF \leftarrow \{\{a_1\}, \{a_2\}, \{e\}\}$, $Min\_G = \{e : \emptyset\}$.
Step 2: Expand CNF by handling $rule_1$
$derived\_atom \leftarrow e$, $flag \leftarrow True$, $Check \leftarrow \emptyset$,
$CNF \leftarrow \{\{a_1\}, \{a_2\}, \{e\}, \{\neg a_1, e\}, \{\neg a_2, e\}\}$,
$Premise \leftarrow \{a_1, a_2\}$, $Min\_G[e] = \{\{a_1, a_2\}\}$.
Step 3: Expand CNF by handling $rule_2$
$derived\_atom \leftarrow e$, $flag \leftarrow True$
$CNF \leftarrow \{\{a_1\}, \{a_2\}, \{e\}, \{\neg a_1, e\}, \{\neg a_2, e\}\}, \{\neg b_1, e\}, \{\neg b_2, e\}\}$
$A \leftarrow \{b_1, b_2\}$, $Premise = \{b_1, b_2\}$,
$Min\_G[e] = \{a_1, a_2\} \times \{b_1, b_2\}$,
$Min\_G[e] = \{\{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\}, \{a_2, b_2\}\}$.
Step 4: Expand CNF by handling $rule_3$
$derived\_atom \leftarrow f$, $flag \leftarrow False$,
$CNF \leftarrow \{\{a_1\}, \{a_2\}, \{e\}, \{\neg a_1, e\}, \{\neg a_2, e\}\}, \{\neg b_1, e\}, \{\neg b_2, e\}, \{\neg c, f\}, \{d, f\}\}$,
$A \leftarrow \{b_1, b_2, c, d, f\}$.
Step 5: Expand CNF by encoding the condition for minimal model generation
$CNF \leftarrow CNF \cup Min\_G[e]$
Step 6: Expand CNF by adding all negative atoms in A.
$CNF \leftarrow CNF \cup \{\{\neg b_1\}, \{\neg b_2\}, \{\neg c\}, \{\neg d\}, \{\neg f\}\}$.
Output: CNF

Example 1 presents how Algorithm 1 generates a satisfiable CNF formula if the generated model is the minimal model. Following Lines 1-10 of Algorithm 1, the first step is initialization. Following Lines 11-29, CNF is extended by handling each rule in $rules$. The expansion process is executed three times as $rules$

contains three rules, presented from Step 2 to Step 4. Following Lines 30-31, CNF is extended by adding all premises of the generated atoms in Step 5. Following Line 32, $Check$ is an empty set, the instructions described in Lines 33-34 are not executed. Following Lines 35-37, CNF is extended by all negative atoms in A in Step 6. The output is $\{\{a_1\},\{a_2\},\{e\},\{\neg a_1,e\},\{\neg a_2,e\}\},\{\neg b_1,e\},\{\neg b_2,e\}$, $\{\neg c,f\},\{d,f\},\{a_1,b_1\},\{a_1,b_2\},\{a_2,b_1\},\{a_2,b_2\},\{\neg b_1\},\{\neg b_2\},\{\neg c\},\{\neg d\},\{\neg f\}\}$. The output is satisfiable when $a_1 \leftarrow True$, $a_2 \leftarrow True$, $b_1 \leftarrow False$, $b_2 \leftarrow False$, $c \leftarrow False$, $d \leftarrow False$, $e \leftarrow True$, and $f \leftarrow False$.

Example 2. Input: $T \cup M$, where $T = rules \cup atoms$,
$rules = \{rule_1\}$, $atoms = \{a_1, a_2\}$, $M = \{a_1, a_2, e, f\}$
$rule_1 = a_1 \wedge a_2 \rightarrow e$
Step 1: Initialization
$G \leftarrow \{e, f\}$, $A \leftarrow \emptyset$, $Check \leftarrow G$,
$CNF \leftarrow \{\{a_1\},\{a_2\},\{e\},\{f\}\}$, $Min\_G = \{e : \emptyset, f : \emptyset\}$.
Step 2: Expand CNF by handling $rule_1$
$derived\_atom \leftarrow e$, $flag \leftarrow True$, $Check \leftarrow \{f\}$,
$CNF \leftarrow \{\{a_1\},\{a_2\},\{e\},\{f\},\{\neg a_1,e\},\{\neg a_2,e\}\}$,
$Premise \leftarrow \{a_1, a_2\}$,
$Min\_G = \{e : \{\{a_1, a_2\}\}, f : \emptyset\}\}$
Step 3: Expand CNF by encoding conditions for minimal model generation
$CNF \leftarrow CNF \cup \{\{a_1, a_2\}\}$.
Step 4: Assign an unsatisfiable formula to CNF.
$CNF \leftarrow \{\{f\},\{\neg f\}\}$
Return CNF

Example 2 presents how Algorithm 1 generates an unsatisfiable CNF formula if the generated model is a model but not the minimal model. Following Lines 1-10 of Algorithm 1, the first step is initialization. Following Lines 11-29, the second step is the extension of CNF by handling $rule_1$. As there is only one rule in $rules$, the expansion process only is executed once. Following Lines 30-31 of Algorithm 1, CNF is extended by encoding the minimal model condition in Step 3. Step 4 follows Lines 32-34. $Check$ contains f, which implies f cannot be derived by the given first-order theory, T. Therefore, CNF is assigned with an unsatisfiable CNF formula: $\{\{f\},\{\neg f\}\}$, which is returned as the output. The CNF formula is inconsistent, therefore, it is unsatisfiable.

Example 3. Input: $T \cup M$, where $T = rules \cup atoms$,
$rules = \{rule_1\}$, $atoms = \{a_1, a_2\}$, $M = \{a_1, a_2\}$
$rule_1 = a_1 \wedge a_2 \rightarrow e$
Step 1: Initialization
$G \leftarrow \emptyset$, $A \leftarrow \emptyset$, $Check \leftarrow \emptyset$, $CNF \leftarrow \{\{a_1\},\{a_2\}\}$, $Min\_G = \emptyset$.
Step 2: Expand CNF by handling $rule_1$
$derived\_atom \leftarrow e$, $flag \leftarrow False$,
$CNF \leftarrow \{\{a_1\},\{a_2\},\{\neg a_1,e\},\{\neg a_2,e\}\}$,
$Premise \leftarrow \{a_1, a_2\}$, $A \leftarrow \{e\}$.

Step 3: Expand CNF by adding all negative atoms in A.
$CNF \leftarrow \{\{a_1\}, \{a_2\}, \{\neg a_1, e\}, \{\neg a_2, e\}, \{\neg e\}\}$

Example 3 presents how Algorithm 1 generates an unsatisfiable CNF formula if the generated model is not a model. Following Lines 1-7 of Algorithm 1, the first step is initialization. Following Lines 11-29, the second step is the extension of CNF by handling $rule_1$. As there is only one rule in $rules$, the expansion process only is executed once. As Min_G is empty, the instructions described in Lines 30-31 are not executed. Step 3 follows Lines 32-37. CNF is extended by adding all negative atoms in A, as $Check$ is empty. The output is $\{\{a_1\}, \{a_2\}, \{\neg a_1, e\}, \{\neg a_2, e\}, \{\neg e\}\}$. The CNF formula is unsatisfiable, because no truth assignment can make it true.

Proposition 1. *Following Definition 4, an input for Algorithm 1 consists of a set of atoms, M, and a first-order theory, T, where T consists of rules and atoms. The output of Algorithm 1 is a CNF formula following Definition 5, denoting as CNF. CNF is satisfiable iff the $atoms \cup M$ is the minimal model of T.*

Proof. We briefly explain how Algorithm 1 generates CNF for the given input. CNF consists of three parts. The first part is the CNF formula that is logically equivalent to T, denoting F_1. The second part is the CNF formula that encodes all possibilities for the derivation of the generated atoms, denoting F_2. The third part is the CNF formula that encodes all negative atoms, denoting F_3.

There are three kinds of relations between $atoms \cup M$ and T. We prove Proposition 1 by cases.
Case 1 $atoms \cup M$ is the minimal model of T.
In this case, F_1 is satisfiable, as there is a model. $F2$ and F_3 are satisfiable, as $atoms \cup M$ is the minimal model of T. F_2 is satisfiable, which indicates each atom in $atoms \cup M$ can be derived by T. F_3 is satisfiable, which implies no more atoms can be derived by T.
Case 2 $atoms \cup M$ is a model of T, but $atoms \cup M$ is not the minimal model of T.
In this case, there is at least one atom that cannot be derived by T. Following Lines 32-34, the output will be an inconsistent CNF formula. Therefore, the output is unsatisfiable.
Case 3 $atoms \cup M$ is not a model of T.
In this case, $\exists c.T \vdash c$, and $c \notin atoms \cup M$. Following Lines 32-37, either an inconsistent CNF formula is generated, or the output contains $\{\neg c\}$. The output is unsatisfiable in both cases.

5 Empirical Analysis

In this section, we analyze the time cost brought by the SAT-Solving component integrated into the *vGOAL* interpreter, as introduced in [16]. To conduct the comparison, we use the same autonomous logistic system case study described

in [16], comprising three autonomous mobile robots denoted as A_1, A_2, and A_3. There are four durative high-level actions, each subject to potential success or failure. Failures are classified as non-fatal or fatal, with non-fatal errors causing the agent to drop its current goal, while fatal errors result in goal redistribution among other agents and removal of the faulty agent.

The *vGOAL* interpreter consistently generates the same decision for a given input. Following Def 1, only agent specifications, MAS, can be different during the executions. Moreover, each agent is specified as $(id, B, goals, M_S, M_R)$. id are not modified during the execution, while $goals$, M_S, and M_R are modified due to the modifications of B. Therefore, it is sufficient to illustrate the time cost brought by the SAT-solving component using three representative experiments: (i) successful completion of all four delivery goals, (ii) successful completion of the first three goals with one non-fatal action failure in the fourth, and (iii) successful completion of the first three goals with one fatal action failure in the fourth.

We conducted these experiments using both versions of the *vGOAL* interpreter, with and without the SAT-solving component. Each run lasted six to eight minutes, with ROS providing real-time sensor updates every 0.5 seconds, resulting in 720 to 960 updates per run for real-time decision-making. All experiments demonstrated safe robot behavior, with no exceptions raised by the SAT-solving component.

Notably, during action execution, sensor updates frequently duplicate previous information. When such duplication occurs, the interpreter does not generate any decisions for the agent. Consequently, the SAT-solving component, which is integrated into the decision-making process, remains inactive. Therefore, minimal time discrepancies were observed between the two versions of the *vGOAL* interpreter during experiments. To precisely measure the time cost introduced by the SAT-solving component, we extracted sensor inputs from the original real-time data, modifying only the occurrence of repeated sensor information.

All experiments are conducted with a MacBook Air 2020 with an Apple M1 and 16GB of RAM. Detailed information regarding the complete *vGOAL* specification of the case study, is available at [14]. Additionally, we have provided three demonstration videos on [14]: an error-free run, a run involving a non-fatal error, and a run involving a fatal error.

Table 4 presents the results of experiments conducted, labeled numerically under the column "Experiment." The "Step" column denotes the step number of the decision-making generation. The "Repeated" column indicates whether repeated sensor information was present. The "System" column specifies the number of agents in the system, while "Active" denotes the number of agents that have goals to achieve. "Decision" indicates the number of decisions made at the step, and "Error" categorizes errors encountered during the experiment: no errors, fatal errors, or non-fatal errors). The columns labeled "T_1(s)" and "T_2(s)" represent the time taken for the *vGOAL* interpreter without or with the SAT-Solving component. The last column, labeled "SAT", denotes the call numbers

Table 4. Experiment Results

Experiment	Step	Repeated	System	Active	Decision	Error	T_1(s)	T_2(s)	SAT
1,2,3	1	No	3	3	2	No	0.82	0.88	352
1,2,3	2	Yes	3	3	0	No	4.60E-5	4.50E-5	0
1,2,3	3	No	3	3	0	No	0.64	0.67	249
1,2,3	4-7	No	3	3	0	No	3.72E-5	3.08E-5	0
1,2,3	8	No	3	3	1	No	0.31	0.33	102
1,2,3	9	No	3	3	1	No	0.38	0.41	151
1,2,3	10	No	3	3	1	No	0.59	0.63	217
1,2,3	11	No	3	3	1	No	0.65	0.71	309
1,2,3	12	No	3	3	0	No	0.61	0.64	192
1,2,3	23	No	3	2	1	No	0.61	0.65	246
1,2,3	24	No	3	2	0	No	0.44	0.46	135
1,2,3	25	No	3	2	1	No	0.47	0.50	165
1,2,3	26	No	3	2	1	No	0.24	0.25	78
1,2,3	27	No	3	2	1	No	0.50	0.54	177
1,2,3	28	No	3	2	0	No	0.44	0.45	105
1,2,3	29	Yes	3	2	0	No	3.79E-5	3.70E-5	0
1,2,3	30	No	3	1	1	No	0.46	0.50	157
1,2,3	31	No	3	1	0	No	0.44	0.46	99
1,2,3	32	Yes	3	1	0	No	3.79E-5	3.91E-5	0
1,2,3	33	No	3	1	1	No	0.48	0.52	182
1,2	34	Yes	3	1	0	No	4.08E-5	4.10E-5	0
3	34	No	2	1	0	Fatal	0.40	0.40	0
1,2	35	No	3	1	0	No	0.42	0.44	129
3	35	No	2	1	1	Fatal	0.38	0.41	166
1	39	No	3	1	1	No	0.21	0.22	54
2	39	No	3	1	1	Non-Fatal	0.48	0.51	158
1	51	No	3	1	1	No	0.47	0.50	156
2	45	No	3	1	1	No	0.47	0.50	156
3	55	No	2	1	1	Fatal	0.46	0.48	134

of the SAT-solving components involved in the decision-making generation at the current step.

The three experiments undergo identical decision-making processes for the initial three delivery goals, encompassing Steps 1 to 33. Thus, their efficiency performances are consolidated when N is 33 or less. From Steps 1 to 22, each autonomous system comprises three agents, with each agent trying to accomplish the delivery goal. Subsequently, from Steps 23 to 29, the autonomous systems still consist of three agents, yet only two agents pursue the delivery goal. From Step 30 to Step 33, only one agent retains the delivery goal.

Divergence in results emerges from Step 34 onwards. In Experiment 3, a fatal error disrupts the agent with the delivery goal at Step 34, leading to a reduction in the autonomous system's agent count to two. Experiment 1 and Experiment

2 exhibit deviation from Step 39, where an encountered non-fatal error affects the delivery agent in Experiment 2.

Notably, the final decision by the *vGOAL* interpreter directs the autonomous system to execute a move action from the waiting point to the destination, occurring at Step 51 for Experiment 1, Step 45 for Experiment 2, and Step 55 for Experiment 3.

The table reveals four key observations. First, there exists an almost positive linear relationship between the time required for SAT solving for each decision-making process and the number of calls made to the SAT-solving component. The more calls for the SAT-solving component, the more time cost is brought by the SAT-solving component. Second, the calls for the SAT-solving component are positively linear to the agent number of the autonomous system, and the number of generated decisions. Third, the *vGOAL* interpreter generates decisions without involving the SAT-solving component when handling repeated sensor information. As shown in Table 4, the calls for the SAT-solving component are zero when the column for "Repeated" is "Yes". Finally, the time cost brought by the SAT-solving component is at most 0.06s for each decision-making generation.

6 Conclusion

This paper presents the integration of an SAT-solving component into the core implementation of the existing *vGOAL* interpreter: the minimal model generation. The SAT-solving component consists of two subcomponents: an SAT encoding component and an SAT solver. Leveraging PySAT for its interface to advanced SAT solvers, our main contributions lie in SAT encoding. We introduce Algorithm 1, effectively encoding the inputs and outputs of the minimal model generation process into a CNF formula, accompanied by a proof sketch to establish its correctness. The empirical results obtained from experiments conducted in an autonomous logistic system illustrate the practical usability and efficiency of the SAT-solving component. The SAT-solving component can significantly enhance the confidence of the *vGOAL* interpreter.

The implementation correctness of the *vGOAL* interpreter is crucial for users to establish trust in the *vGOAL* approach. We face four challenges: complexity, guaranteed assurance, efficiency, and automation. The SAT-solving integration offers a practical solution by focusing on the error-prone minimal model generation, verifying the generated results rather than the entire process, and leveraging PySAT within the existing *vGOAL* interpreter.

Finally, we briefly discuss the potential applicability of the SAT-solving integration to other APL interpreters. Belief-Desire-Intention (BDI) APLs are the most popular paradigm of APLs encompassing BDI reasoning [3], which typically involves logical derivation. To show the correctness of the logical derivation component, we need to show every derived atom can be derived from the first-order theory, as outlined in Algorithm 1. Specifically, Algorithm 1 can be used to encode the logical derivation component by removing the encoding part from Lines 35-37.

Acknowledgements

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

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. Cardoso, R.C., Ferrando, A.: A review of agent-based programming for multi-agent systems. Computers **10**(2), 16 (2021)
4. 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)
5. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Automated software engineering **19**(1), 5–63 (2012)
6. Gong, W., Zhou, X.: A survey of sat solver. In: AIP Conference Proceedings. vol. 1836. AIP Publishing (2017)
7. Hindriks, K.V.: Programming rational agents in GOAL. In: Multi-agent programming, pp. 119–157. Springer (2009)
8. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: A Python toolkit for prototyping with SAT oracles. In: SAT. pp. 428–437 (2018)
9. Jongmans, S.S.T., Hindriks, K.V., Van Riemsdijk, M.B.: 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. pp. 219–237. Springer (2010)
10. Marques-Silva, J.: Practical applications of boolean satisfiability. In: 2008 9th International Workshop on Discrete Event Systems. pp. 74–80. IEEE (2008)
11. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535 (2001)
12. Shoham, Y.: Agent-oriented programming. Artificial intelligence **60**(1), 51–92 (1993)
13. Weiss, G.: Multiagent Systems. The MIT Press (2013)
14. Yang, Y.: Supplementary Documents. `https://drive.google.com/drive/folders/16xXEqjg41GWF2zMeR2phpqCuiCKo8m5L?usp=share_link` (2024)
15. Yang, Y., Holvoet, T.: Making model checking feasible for goal. Annals of Mathematics and Artificial Intelligence pp. 1–17 (2023)
16. Yang, Y., Holvoet, T.: Safe autonomous decision-making with vGOAL. In: Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics. The PAAMS Collection. Guimarães, Portugal (7 2023)
17. Yang, Y., Holvoet, T.: vgoal: A goal-based specification language for safe autonomous decision-making. In: Ciortea, A., Dastani, M., Luo, J. (eds.) Engineering Multi-Agent Systems - 11th International Workshop, EMAS 2023, London, UK, May 29-30, 2023, Revised Selected Papers. Lecture Notes in Computer Science, vol. 14378, pp. 41–58. Springer (2023)