Towards Verifying a Blocks World for Teams GOAL Agent

Alexander Birch Jensen[a]
DTU Compute, Department of Applied Mathematics and Computer Science, Technical University of Denmark, Richard Petersens Plads, Building 324, DK-2800 Kongens Lyngby, Denmark

Keywords: Agent Programming, Formal Verification, Agent Logic, GOAL.

Abstract: We continue to see an increase in applications based on multi-agent system technology. As the technology becomes more widespread, so does the requirement for agent systems to operate reliably. In this paper, we expand on the approach of using an agents logic to prove properties of agents. Our work describes a transformation from GOAL program code to an agent logic. We apply it to a Blocks World for Teams agent and prove a correctness property. Finally, we sketch future challenges of extending the framework.

1 INTRODUCTION

It is of key importance to provide assurance of the reliability of multi-agent systems (Dix et al., 2019). However, demonstrating the reliability of such systems remains a challenge due to their often complex behavior, usually exceeding the complexity of procedural programs (Winikoff and Cranefield, 2014).

Popular approaches to programming agents have been inspired by the agent-oriented programming paradigm (Shoham, 1993). Examples include JADE for the Java-platform and GOAL that is inspired by logic programming (Bellifemine et al., 2001; Hindriks et al., 2001). A notable difference between the two is that GOAL implements a BDI model (Rao and Georgeff, 1991) whereas JADE is a middleware that facilitates development of multi-agent systems under the FIPA standard (Poslad, 2007). In this paper, we will focus our efforts towards the GOAL agent programming language as its compact formal semantics gives a solid foundation for the development of verification mechanisms.

An approach to verifying agents GOAL is proposed in (de Boer et al., 2007) using a verification framework. The framework is a complete programming theory for GOAL — it gives the semantics of the GOAL language and a temporal logic proof theory for proving properties of agents. We extend this approach by describing a transformation from program code to the agent logic.

We apply the verification framework to an instance of a Blocks World for Teams (BW4T) problem (Johnson et al., 2009). We start from program code and transform it to an agent logic from which we prove its correctness.

The paper is structured as follows. Section 2 relates to existing work in the literature. Section 3 introduces GOAL and the BW4T environment. Section 4 describes a proof theory for GOAL. Section 5 presents a transformation method from GOAL programs to agent logic. Section 6 describes how to verify agents in the agent logic. Finally, Section 7 gives concluding remarks.

2 RELATED WORK

The work in this paper relates to our work on formalizing GOAL and the verification framework in a proof assistant (Jensen, 2021). Unlike this paper, it does not consider the practical aspects of programming the agents and transforming program code to an agent logic. Instead, the paper explores the use of a proof assistant as a tool for verification of agents based on an agent logic. In (Jensen et al., 2021), we consider how a theorem proving approach can contribute to demonstrating reliability for cognitive agent-oriented programming.

The verification framework notation and theory have some resemblance to Misra's UNITY (Misra, 1994), in particular the *ensures* operator. However, a notable difference is that we are working towards verifying a cognitive agent language whereas UNITY is for verification of general parallel programs. The UNITY framework has been mecha-

[a] https://orcid.org/0000-0002-7298-2133

Jensen, A.
Towards Verifying a Blocks World for Teams GOAL Agent.
DOI: 10.5220/0010268303370344
In *Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021) - Volume 1*, pages 337-344
ISBN: 978-989-758-484-8

nized in the higher-order logic theorem prover Isabelle/HOL (Nipkow et al., 2002; Paulson, 2000).

In (Alechina et al., 2010) theorem-proving techniques are applied to verify correctness properties of simpleAPL agents (simpleAPL is a simplified version of 3APL (Hindriks et al., 1999)). In this work agents execute actions based on plans and planning rules whereas GOAL agents decide their next action in each state based on decision rules.

A different approach to verifying agent system is through model-checking where one attempts to verify desired properties over possible states of the system. A propositional logic variant of the verification framework, as we are considering here, is in many ways similar to model-checking: we explore and prove formulas about the possible states of the agent program. However, once we start to consider possible ways to generalize the theory to higher-order logics, we could see some advantages over finite-state models; although this remains speculative and it is not yet clear how this can be achieved. (Dennis and Fisher, 2009) has worked towards techniques for analyzing implemented multi-agent system platforms based on BDI. With a starting point in model-checking techniques for AgentSpeak (Bordini et al., 2006), these are extended to other languages including GOAL. From a recent review of logic-based approaches for multi-agent systems by (Calegari et al., 2020), it is clear that model-checking has retained its position as the primary tool for practical verification of multi-agent systems.

3 BACKGROUND

This section briefly introduces the GOAL programming language and the BW4T environment.

3.1 GOAL Programming Language

Below we give a brief introduction to the GOAL programming language; for further details we direct the reader to (Hindriks, 2009; de Boer et al., 2007).

A multi-agent system (MAS) in GOAL is specified by a configuration file (**.mas2g*). It specifies the environment to connect to, if any; and which agents to start and when. The agent specification may state a number of modules (**.mod2g*) to be used by the agent. An initialization module may be used to set up the initial beliefs and goals of the agent. The belief and goal base constitutes the agent's mental state. An event module may be used to update the agent's mental state, usually also based on environment perception. Lastly, a main module specifies decision rules for action selection by a number of rules of the form `if` **`condition`** `then` **`action`**. The condition is over the mental state of the agent where `bel(query)` performs a Prolog-like query on the belief base. Likewise, `goal(query)` queries the goal base. The available actions with the pre- and postconditions are specified in an action specification *(*.act2g)*; usually the available actions are made available by the environment, but internal actions may be defined. Each module may specify knowledge bases, most commonly Prolog files *(*.pl)*, that specify available belief predicates and usually also auxiliary functions.

We now turn to the semantics of the language. A rule is applicable if its condition holds and is then said to be enabled; its precondition should also be met as defined in the action specification. The default behavior is to select the first applicable rule. The execution is performed in cycles as follows:

1. Process any new events.
2. Select an action based on the decision rules.
3. Perform the selected action.
4. Update state based on pre- and postconditions.

As such, (1) is associated with the event module and processing of received messages from the environment, (2) with the main module; (3) sends a request to the environment and (4) updates the agent's mental state based on the action specification. GOAL applies the blind commitment strategy in which a goal is only dropped when it has been completely achieved (Rao and Georgeff, 1993).

3.2 The BW4T Environment

In the BW4T environment, the goal is to collect blocks located in different rooms in a particular sequence of colors. Pathfinding between rooms is handled by the environment server. For now, we restrict ourselves to a single-agent instance. Table 1 shows the relevant subset of the available percepts and actions of the BW4T environment.

Following the Russell & Norvig categorization of environments (Russell and Norvig, 2020), BW4T can for our configuration be categorized as a single-agent, partially observable, deterministic environment. This categorization plays an essential part in enabling modelling of the environment in the verification framework. We will assume the environment to be fully observable which in turn is ensured by running on a predefined map.

We have designed a simple, deterministic custom map that will be our example instance. Figure 1 shows the initial state of the example map where the

Table 1: Relevant percepts and actions for our agent in the BW4T environment.

Percepts	
`in(RoomId)`	The agent is in the room `<RoomId>`.
`atBlock(BlockId)`	The agent is at the block `<BlockId>`.
`holding(BlockId)`	The agent is holding the block `<BlockId>`.
Actions	
`goTo(RoomId)`	The agent moves to the room `<RoomId>`.
`goToBlock(BlockId)`	The agent moves to the block `<BlockId>` in the current room.
`pickUp(BlockId)`	The agent picks up the block `<BlockId>`.
`putDown`	The agent puts down the block it is carrying.

goal is to collect a single red block (indicated in the bottom left corner).

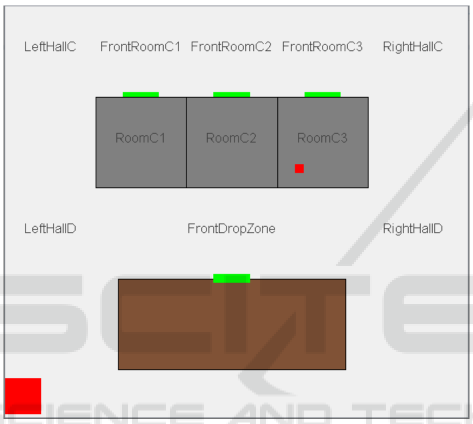

Figure 1: The example map in the initial state.

3.3 Distributed Files

This section describes important details about the implementation of the BW4T agent in GOAL.

The source code is publicly available online:

https://people.compute.dtu.dk/aleje/public/

To run the program, the BW4T map file should be copied to the *maps* subfolder in a BW4T environment installation. It will then become available in the map selector when running the environment server. If using Eclipse for GOAL[1], the remaining files can be imported to a new example BW4T project.

The MAS file connects to the environment and creates a single agent entity. The initialization sets up the agent's initial mental state. It is not inherently possible for agents to perceive block positions before entering rooms and is thus hardcoded to ensure full observability. The event module updates the agent's belief base in each execution cycle with regard to relevant percepts. The main module states the decision rules used by the agent to select an appropriate action in each cycle. The knowledge base is a Prolog file that in our case simply defines the belief predicates and their arities. Finally, the BW4T map file is a custom map configuration that sets up the example situation used throughout this paper.

[1]https://goalapl.atlassian.net/wiki/

4 PROOF THEORY FOR GOAL

A temporal logic is constructed on top of GOAL. It differs from regular definitions of temporal logic by having its semantics derived from the semantics of GOAL and by incorporating the belief and goal operators. Hoare triples are used to specify effect of actions on mental states. A Hoare triple $\{\varphi\}\ \rho \triangleright do(a)\ \{\psi\}$ for a conditional action a, where the truth of ρ determines if a is enabled, holds if φ is true in the mental state before execution of a and ψ is true in the mental state following the execution of a.

4.1 Basic Action Theory

A basic action theory specifies the effects of agent capabilities by associating Hoare triples with effect axioms. Effect axioms are Hoare triples that state the effects of actions. By means of an *enabled* predicate for each action, the conditions on actions are also specified. Additionally, the theory specifies what *does not change* following execution of actions by associating Hoare triples with frame axioms. Effect and frame axioms should be specified by the user — in our case effect axioms are the result of the program code transformation as described in Section 5. It may also be useful to prove invariants of the program help with proofs. Frame axioms and invariants have a close relation: frame axioms express stable properties and invariants are stable properties that also hold initially.

4.2 Proof System

We now consider a Hoare system for GOAL. The proof rules are as defined in (de Boer et al., 2007) and are listed in Table 2.

The rule for infeasible actions allows derivation of frame axioms for actions on mental states in which they are not enabled. The rule for conditional actions allows us to prove a Hoare triple for a conditional action from that of a basic action. We use the notion basic action when referring to an action but not its decision rule. The three remaining rules are structural rules that allow us to combine Hoare triples (the conjunction and disjunction rule) and strengthening the precondition and weakening the postcondition (the consequence rule).

For further studies of the Hoare system, we direct the reader to Section 4 in (de Boer et al., 2007) in which the Hoare system is proved sound and complete with respect to the semantics of Hoare triples on mental state formulas.

5 TRANSFORMATION

In order to make the link between GOAL program code and the verification framework, we describe how to perform a mechanical transformation from GOAL code to the agent logic. Due to current limitations of the framework, we have to assume the following:

(1) Single agent assumption: only the agent executes actions.

(2) The environment is deterministic: an action has a deterministic outcome.

(3) The agent has complete knowledge about the initial state of the environment and its initial beliefs and goals are completely specified.

(4) Action execution is instantaneous (no durative actions).

Assuming a single agent to be the only actor in the environment plays into the fact that the environment is not modelled in the framework. Percept handling and environment interactions is a general issue in verification frameworks that has not been addressed effectively: any non-agent interaction must be integrated into the agent's beliefs and capabilities. For instance, in the BW4T environment, an agent is not initially aware of the position of blocks; the agent becomes aware only when it enters the room in which the blocks are located. It is not clear how to deal with such interactions: we need to apply some level of domain knowledge to integrate this interaction; in our case the simplest approach is to add the block locations to the agent's knowledge. A similar problem occurs when dealing with non-determinism in the environment: we have no way of dealing with dynamic changes, not necessarily known to the agent, and it is not clear how to model this either.

In order to make the transformation as mechanical as possible, we impose restrictions on the structure of the GOAL code:

(1) The main module (for decision-based action selection) should only have rules of the form `if` φ `then` a where φ is a query to the mental state and a is an action in the environment.

(2) The preconditions of an action should specify the minimal condition under which the action is enabled (action specification).

(3) Rules for environment interaction should be annotated with an action name (event module).

Let us elaborate on these restrictions. The restrictions on the form of the action decision rules are simply in place to allow a straightforward translation that plays well with the specification of Hoare triples. A minimal action specification is at the core of the philosophy of GOAL: changes to the mental state should be based on perceived changes in the environment whenever possible. As such, this is more of a recommendation than a restriction that also eases the translation process. Lastly, the annotation of code for environment interaction in the event module is to make the process as mechanical as possible — recall that events are handled before the main module in each cycle. We need to translate environment interaction to the pre- and postconditions of actions. Thus, we need to be able to associate these condition with an action specification.

5.1 BW4T Transformation

We now apply the transformation method to the *goTo* action for our BW4T agent.

Hoare triples are derived from the GOAL code by transforming the relevant parts of the action specification and the event and main module. To avoid branching in the proof shown later, we have ensured that decision rules are mutually exclusive.

Consider first the action specification `goTo`:

```
define goTo(X) with
    pre { true }
    post { true }
```

The effects of actions are perceived through changes in the environment and are left empty (simply *true*). The precondition specifies the minimal conditions for

Table 2: The proof rules of our Hoare system.

Infeasible actions:	$\dfrac{\varphi \longrightarrow \neg enabled(a)}{\{\varphi\}\ a\ \{\varphi\}}$
Rule for conditional actions:	$\dfrac{\{\varphi \wedge \psi\}\ a\ \{\varphi'\} \qquad (\varphi \wedge \neg\psi) \longrightarrow \varphi'}{\{\varphi\}\ \psi \triangleright do(a)\ \{\varphi'\}}$
Consequence rule:	$\dfrac{\varphi' \longrightarrow \varphi \qquad \{\varphi\}\ a\ \{\psi\} \qquad \psi \longrightarrow \psi'}{\{\varphi'\}\ a\ \{\psi'\}}$
Conjunction rule:	$\dfrac{\{\varphi_1\}\ a\ \{\psi_1\} \qquad \{\varphi_2\}\ a\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ a\ \{\psi_1 \wedge \psi_2\}}$
Disjunction rule:	$\dfrac{\{\varphi_1\}\ a\ \{\psi\} \qquad \{\varphi_2\}\ a\ \{\psi\}}{\{\varphi_1 \vee \varphi_2\}\ a\ \{\psi\}}$

action execution — here, it is always possible to go to another room. This yields the following skeleton for our final Hoare triple:

$$\{true\}\ enabled(goTo(X)) \triangleright do(goTo(X))\ \{true\}$$

Note that *enabled* is not yet specified.

Below is the code annotated with *goTo* from the event module:

```
% act-goTo
if bel(in(X)), not(percept(in(X))) then
    delete(in(X)).
if percept(in(X)), not(bel(in(X))) then
    insert(in(X)).
```

Since the agent has full control, we can model the changes to the environment brought about by actions by mapping code in the event module to pre- and post-conditions of actions. The annotation `act-goTo` tells us that it is related to the *goTo* action. The first rule has the pattern `bel(Q), not(percept(Q))` used for removing outdated beliefs. The second rule pattern inserts a new belief, thus yielding:

$$\{\mathsf{B}(in(Y) \wedge \neg in(X))\}$$
$$enabled(goTo(X)) \triangleright do(goTo(X))$$
$$\{\mathsf{B}(in(X) \wedge \neg in(Y))\}$$

Since $\mathsf{B}(in(X) \wedge \neg in(X))$ is never true, we implicitly ensure that $X \neq Y$ holds. The final step of deriving the Hoare triple is to transform code from the main module to the specification of *enabled*(*goTo*(X)). That is, the condition under which the action should be enabled for the agent. The decision rule for *goTo* is as follows:

```
if goal(collect(C)),
    not(holding(_)), color(_, C, X)),
    not(bel(in(Y), color(_, C, Y)))
        then goTo(X).
```

GOAL allows use of _ for variables we do not care about. We cannot do this in our logic so we introduce variable symbols:

$$enabled(goTo(X)) \longleftrightarrow \mathsf{G}(collect(C)) \wedge$$
$$\mathsf{B}(\neg holding(U) \wedge color(V,C,X)))$$
$$\wedge \neg \mathsf{B}(in(Y) \wedge color(W,C,Y))$$

Before we are done, the Hoare triple should be instantiated with propositional symbols. Consider the following initial state (capturing the initial state of the example map):

$$\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \mathsf{G}collect(red)$$

In the above, r_0 is the initial position of the agent (the drop zone).

The derived Hoare triples must be instantiated with propositional symbols. Below is the effect axiom for *goTo* for going from r_0 to r_1:

$$\{\mathsf{B}(in(r_0) \wedge \neg in(r_1))\}\ \mathsf{G}(collect(red)) \wedge$$
$$\mathsf{B}(\neg holding(b_a) \wedge color(b_a, red, r_1)) \wedge$$
$$\neg \mathsf{B}(in(r_0) \wedge color(b_a, red, r_0)) \triangleright do(goTo(r_1))$$
$$\{\mathsf{B}(in(r_1) \wedge \neg in(r_0))\}$$

The shown instantiation of the effect axiom will be useful for the correctness proof we go through in Section 6. While not apparent from the example, we have a special case for the *goTo* action when going back to the the drop zone r_0. The agent only needs to go to the drop zone to put down blocks. Putting down a (correct) block in the drop zone means the environment considers the block to be collected. Thus, we only show the *enabled* equivalence for *goTo*(r_0):

$$enabled(goTo(r_0)) \longleftrightarrow \mathsf{B}(holding(U) \wedge \neg in(r_0))$$

We perform similar transformations for every action. We are still considering ways to fully automate the process, potentially by imposing additional restrictions on the structure of the GOAL code.

5.2 Automating the Transformation

The transformation method is not yet something that can be fully automated. In this section, we expand on why this is the case, and how it can be fully automated.

One of the main issues is that the verification framework abstracts from a core feature of GOAL programming: interacting with the environment. In particular, the processing of events is problematic. The verification framework assumes that only the actions of agents may change the state of the environment. While this may be feasible for our use of the BW4T environment, the proper approach is to perceive changes in the environment. In order to map the expected effect of performing an action, we need to be able to associate event processing with actions directly. This requires some domain knowledge from the programmer and should therefore be specified. Alleviating this will likely require that the verification framework is able to model the environment. We still need to be able to state some model of the environment, i.e. what are the expected results of actions — if we know that the environment provides an action for moving the agent, but we have no idea what it does, then we are stuck.

Another issue is the assumption of full observability for the agent. This causes us to push domain knowledge to the initial mental state of the agent. This issue very much ties into the issue of not modelling the environment. Also here, the solution seems to be to push the domain knowledge into a model of the environment and have the agent perceive it.

6 PROVING CORRECTNESS

The proof of correctness consists of proofs of **ensures** formulas. The operator is defined as:

$$\varphi\, \mathbf{ensures}\, \psi := (\varphi \longrightarrow (\varphi\, \mathbf{until}\, \psi)) \wedge (\varphi \longrightarrow \Diamond\psi)$$

Informally, φ **ensures** ψ means that φ guarantees the realization of ψ. In sequence they prove that the agent reaches its goal in a finite number of steps.

The operator $\varphi \mapsto \psi$ is the transitive, disjunctive closure of **ensures**:

$$\frac{\varphi\ \mathbf{ensures}\ \psi}{\varphi \mapsto \psi} \qquad \frac{\varphi \mapsto \chi \quad \chi \mapsto \psi}{\varphi \mapsto \psi}$$

$$\frac{\varphi_1 \mapsto \psi \ \dots\ \varphi_n \mapsto \psi}{(\varphi_1 \vee \dots \vee \varphi_n) \mapsto \psi}$$

The operator differs from **ensures** as φ is not required to remain true until ψ is realized.

The proof of a formula φ **ensures** ψ requires that we show that every action a satisfies the Hoare triple $\{\varphi \wedge \neg\psi\}\ a\ \{\varphi \vee \psi\}$ and that there is at least one action a' which satisfies the Hoare triple $\{\varphi \wedge \neg\psi\}\ a'\ \{\psi\}$.

6.1 BW4T Agent Correctness Proof

In order to prove the correctness of an agent, we need to show that it reaches a desired state from its initial state. Using the operator defined above, this means that we need to prove $\varphi \mapsto \psi$ where φ is the initial state and ψ is the desired state. As stated, this can be split up into a number of **ensures** proof steps where we consider each intermediate state and prove the transitions between them.

It is often useful to expand our base of derived Hoare triples yielded by the transformation. Proving program invariants that are useful in multiple steps of the proof is an effective strategy to minimize the workload. We claim the following invariant *inv-in* to hold:

$$\mathsf{B}\left(\forall r, r'((in(r) \wedge r \neq r') \longrightarrow \neg in(r'))\right)$$

The invariant states that the agent can only be in one place at any point in time. Invariants must be proved in the proof theory — due to space we do not show the proof here.

We find that the correctness property for the BW4T agent is: from its initial state, the agent must reach a state where it believes the red block to be collected:

$$\begin{aligned}&\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \\ &\mathsf{G}collect(red) \mapsto \mathsf{B}collect(red)\end{aligned}$$

The full proof outline is a sequence of **ensures** statements that lead to the desired mental state as shown in Figure 2. This sequence is explicitly stated and of course ties into the belief that there exist actions that will satisfy each step.

In the proof outline in Figure 2 the changes to the mental state in each step are underlined. The proof outline consists of 5 steps. Each step indicates the transition from one mental state to another caused by the execution of an action. The fact that our decision rules for actions are mutually exclusive makes each step easier to prove: at each of the five steps only a single action is enabled. In other words we only need to consider a single execution trace.

The proof is completed by proving each of the 5 steps. If we consider step (1) this means that we need

(1) $\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)$
ensures
$\mathsf{B}color(b_a, red, r_1) \wedge \underline{\mathsf{B}in(r_1)} \wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)$

(2) $\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1) \wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)$
ensures
$\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1) \wedge \neg\mathsf{B}holding(b_a) \wedge \underline{\mathsf{B}atBlock(b_a)} \wedge \mathsf{G}collect(red)$

(3) $\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1) \wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{B}atBlock(b_a) \wedge \mathsf{G}collect(red)$
ensures
$\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1) \wedge \underline{\mathsf{B}holding(b_a)} \wedge \mathsf{G}collect(red)$

(4) $\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1) \wedge \mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)$
ensures
$\mathsf{B}color(b_a, red, r_1) \wedge \underline{\mathsf{B}in(r_0)} \wedge \mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)$

(5) $\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)$
ensures
$\underline{\mathsf{B}collect(red)}$

Figure 2: Proof outline consisting of five steps.

to prove, for every action a:

$\{\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \neg\mathsf{B}holding(b_a) \wedge$
$\mathsf{G}collect(red) \wedge \neg(\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1)$
$\wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red))\}$
a
$\{\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \neg\mathsf{B}holding(b_a) \wedge$
$\mathsf{G}collect(red) \vee \mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1)$
$\wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)\}$

For at least one action a', in our case $goTo(r_1)$, we are further required to prove:

$\{\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \neg\mathsf{B}holding(b_a) \wedge$
$\mathsf{G}collect(red) \wedge \neg(\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1)$
$\wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red))\}$
a'
$\{\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_1) \wedge \neg\mathsf{B}holding(b_a) \wedge$
$\mathsf{G}collect(red)\}$

Each of these Hoare triples are proved by applying the proof rules of Table 2. The effect axioms are derived from the described transformation method while the frame axioms, describing what does not change, as of now have to be supplied manually by the user. Due to space limitations example derivations are left out; they may be found online as a separate appendix file (see Section 3.3).

7 CONCLUSION

We have presented an approach for verification of an implemented GOAL agent program for Blocks World for Teams using a verification framework. We have described a method for transforming GOAL program code into expressions of an agent logic for which a temporal logic proof theory is used to prove properties of agents. We exemplified the process from code to proof using a simple BW4T problem.

Reflecting on the agent logic approach, a notable characteristic of agent logics is their close ties to agent programming. In our case, the logic is directly linked to implemented GOAL program code. On the quest to make agent verification more approachable for programmers of agent systems, closing the gap between practice and theory is of key importance. In our future work with the framework there are some key challenges to address. We have only experimented with rather limited scenarios and it will be interesting to see how well it scales for more complex scenarios. We need to consider means to relax the constraint that the agent must have complete knowledge; this constraint severely hinders the usability of the approach. Furthermore, we need to be able to model environments in the framework. Further challenges appear due to our rather basic notion of actions; this notion ought to be extended to also consider non-determinism and real-time constraints (e.g. an action must be fully executed before a deadline). Lastly, we have of course the challenge of modelling systems with more than just a single agent. There have been efforts towards agent logics for multiple agents, such as by (Bulling and Hindriks, 2009), but the the logical languages are rather limited.

We observe that the current state-of-the-art is to apply model-checking tools for verification of agent software (Luckcuck et al., 2019). Efforts towards enhancing the practicality of tools for verifying agents systems are ongoing. We believe that approaches using agent logics, with further work, have potential and could offer an alternative to model-checking approaches. Achieving this will require further research, in particular towards enhancing practicalities.

ACKNOWLEDGEMENTS

I would like to thank Koen Hindriks for discussions and comments on drafts of this paper. I would also like to thank Asta H. From and Jrgen Villadsen for comments on drafts of this paper.

REFERENCES

Alechina, N., Dastani, M., Khan, A. F., Logan, B., and Meyer, J.-J. (2010). *Using Theorem Proving to Verify Properties of Agent Programs*, pages 1–33. Springer.

Bellifemine, F., Poggi, A., and Rimassa, G. (2001). JADE: a FIPA2000 compliant agent development environment. pages 216–217.

Bordini, R. H., Fisher, M., Visser, W., and Wooldridge, M. (2006). Verifying Multi-agent Programs by Model Checking. *Autonomous Agents and Multi-Agent Systems*, 12:239–256.

Bulling, N. and Hindriks, K. V. (2009). Towards a Verification Framework for Communicating Rational Agents. *MATES*, pages 177–182.

Calegari, R., Ciatto, G., Mascardi, V., and Omicini, A. (2020). Logic-based technologies for multi-agent systems: a systematic literature review. *Autonomous Agents and Multi-Agent Systems*, 35.

de Boer, F. S., Hindriks, K. V., Hoek, W., and Meyer, J.-J. (2007). A verification framework for agent programming with declarative goals. *Journal of Applied Logic*, 5:277–302.

Dennis, L. A. and Fisher, M. (2009). Programming Verifiable Heterogeneous Agent Systems. In *Programming Multi-Agent Systems*, pages 40–55. Springer.

Dix, J., Logan, B., and Winikoff, M. (2019). Engineering Reliable Multiagent Systems (Dagstuhl Seminar 19112). *Dagstuhl Reports*, 9(3):52–63.

Hindriks, K. V. (2009). *Programming Rational Agents in GOAL*, pages 119–157.

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

Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. (2001). Agent Programming with Declarative Goals. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 228–243. Springer.

Jensen, A. B. (2021). Towards Verifying GOAL Agents in Isabelle/HOL. In *ICAART 2021 - Proceedings of the 13th International Conference on Agents and Artificial Intelligence*. SciTePress. To appear.

Jensen, A. B., Hindriks, K. V., and Villadsen, J. (2021). On Using Theorem Proving for Cognitive Agent-Oriented Programming. In *ICAART 2021 - Proceedings of the 13th International Conference on Agents and Artificial Intelligence*. SciTePress. To appear.

Johnson, M., Jonker, C., Riemsdijk, B., Feltovich, P. J., and Bradshaw, J. (2009). Joint Activity Testbed: Blocks World for Teams (BW4T). *ESAW*, pages 254–256.

Luckcuck, M., Farrell, M., Dennis, L. A., Dixon, C., and Fisher, M. (2019). Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Comput. Surv.*, 52(5).

Misra, J. (1994). A Logic for Concurrent Programming. Technical report, Formal Aspects of Computing.

Nipkow, T., Paulson, L., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer.

Paulson, L. (2000). Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 1(1):3–32.

Poslad, S. (2007). Specifying Protocols for Multi-Agent Systems Interaction. *TAAS*, 2.

Rao, A. S. and Georgeff, M. P. (1991). Modeling Rational Agents within a BDI-Architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, KR'91, pages 473–484. Morgan Kaufmann Publishers Inc.

Rao, A. S. and Georgeff, M. P. (1993). Intentions and Rational Commitment. In *Proceedings of the First Pacific Rim Conference on Artificial Intelligence (PRICAI-90)*. Citeseer.

Russell, S. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.

Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92.

Winikoff, M. and Cranefield, S. (2014). On the Testability of BDI Agent Systems. *Journal of Artificial Intelligence Research*, 51:71–131.