PhD Thesis

Verification of Cognitive Agent-Oriented Programming in the Isabelle Proof Assistant

Alexander Birch Jensen

Technical University of Denmark

February 2022
ISSN 0909-3192

Abstract

It is of key importance that we are be able to demonstrate reliability of software systems. The highest level of reliability is achieved by means of formal verification. This thesis is concerned with the engineering of reliable cognitive multi-agent systems. A multi-agent system is a software system composed of multiple intelligent agents able to solve complex problems. Agents in a cognitive multi-agent system are further able to exhibit cognitive behavior. It is notoriously difficult to demonstrate reliability for cognitive multi-agent systems because of their complex behavior. Dedicated agent programming languages such as GOAL provide frameworks for programming cognitive agents. For agent programming languages based on a formal semantics it is possible to apply formal verification techniques to demonstrate their reliability. A proof assistant, such as Isabelle/HOL, is a tool that can assist in proving theorems about a formally specified software system. They have proven useful in verifying traditional software systems, but their usefulness has not been explored in verification of intelligent agents. This thesis explores a new approach to formal verification of GOAL agents using the Isabelle/HOL proof assistant. We develop a complex multi-agent system in GOAL that competes at a research-focused competition for multi-agent systems. We then consider existing work on a verification framework for GOAL agents, and develop a method of transformation from specification of GOAL program code to GOAL programming logic. We further consider how the use of proof assistants can contribute to demonstrating reliability for multi-agent systems. We apply our own approach, and formalize said verification framework for GOAL agents in the Isabelle/HOL proof assistant. We apply the formalized verification framework to prove the correctness of an example agent. Finally, we reflect on the contributions, limitations and perspectives of our work from a desire to verify multi-agent systems such as the one we developed for the competition.

Resumé

Det er af afgørende betydning, at vi er i stand til at demonstrere pålidelighed af softwaresystemer. Det højeste niveau af pålidelighed opnås ved hjælp af formel verifikation. Afhandlingen omhandler udvikling af pålidelige kognitive multiagentsystemer. Et multiagentsystem er et softwaresystem bestående af flere intelligente agenter, der er i stand til at løse komplekse problemer. Agenter i et kognitivt multiagentsystem er yderligere i stand til at udvise kognitiv adfærd. Det er notorisk svært at påvise pålidelighed for kognitive multiagentsystemer på grund af deres komplekse adfærd. Dedikerede agentprogrammeringssprog såsom GOAL giver rammer for programmering af sådanne kognitive agenter. For agentprogrammeringssprog baseret på en formel semantik er det muligt at anvende formelle verifikationsteknikker til at demonstrere deres pålidelighed. En bevisassistent, såsom Isabelle/HOL, er et værktøj, der kan hjælpe med at bevise egenskaber af et formelt specificeret softwaresystem. De har vist sig nyttige til verifikation af mere traditionelle softwaresystemer, men deres anvendelighed til verifikation af intelligente agenter er ikke blevet undersøgt. Denne afhandling udforsker en ny tilgang til formel verifikation af GOAL-agenter ved hjælp af Isabelle/HOL-bevisassistenten. Vi udvikler et komplekst multiagentsystem i GOAL, der konkurrerer i en forskningsorienteret konkurrence for multiagentsystemer. Derefter betragter vi eksisterende arbejde med en verifikationsramme for GOAL-agenter og udvikler en metode til transformation fra specifikation af GOAL-programkode til GOAL-programmeringslogik. Vi betragter yderligere hvordan brugen af bevisassistenter kan bidrage til at demonstrere pålidelighed af multiagentsystemer. Vi anvender vores egen tilgang ved at formalisere den nævnte verifikationsramme for GOAL-agenter i Isabelle/HOL-bevisassistenten. Vi bruger den formaliserede verifikationsramme til at bevise korrektheden af en eksempelagent. Til sidst reflekterer vi over bidrag, begrænsninger og perspektiver i vores arbejde ud fra et ønske om at verificere multiagentsystemer som det, vi udviklede til konkurrencen.

Preface

This thesis is the result of work on my PhD studies from 1 February 2019 to 14 February 2022 (DTU PhD Scholarship for 3 years with 2 weeks extension due to parental leave).

The PhD studies were conducted at the Section for Algorithms, Logic and Graphs (AlgoLoG) at DTU Compute. The studies were supervised by Associate Professor Jørgen Villadsen and co-supervised by Associate Professor Sebastian Mödersheim.

Part of the studies took place abroad (just before the COVID-19 pandemic). I attended the Dagstuhl Seminar "Engineering Reliable Multiagent Systems" (event 19112) organized by Jürgen Dix, Brian Logan and Michael Winikoff from 10 March 2019 to 15 March 2019. I visited Associate Professor Jasmin Blanchette and Full Professor Koen Hindriks at Vrije Universiteit Amsterdam from 17 June 2019 to 26 July 2019. I went to the University of Latvia, Riga, for the 31st European Summer School In Logic, Language and Information (ESSLLI) from 5 August 2019 to 16 August 2019.

Acknowledgements

Koen Hindriks has discussed with me the topics of this thesis during my visit to Amsterdam and later in online meetings. These discussions were essential for the results presented in this thesis. Jørgen Villadsen and Sebastian Mödersheim have flawlessly supervised my PhD studies. Jasmin Blanchette has shown great hospitality during my visit to Amsterdam. Anders Schlichtkrull, Asta Halkjær From, Frederik Krogsdal Jacobsen and John Bruntse Larsen have commented on drafts of various papers. Lastly, Asta Halkjær From and Frederik Krogsdal Jacobsen have commented on a draft of this thesis.

Contents

Chapter 1

Introduction

One of the pillars of software development is the ability to deliver and deploy reliable software systems. By a reliable system we mean a system that works according to its specification and is free from errors. Enormous effort is put into ensuring that programs operate reliably. The most accessible methods of ensuring reliability are those based on testing where, either automatically or manually, it is tested that a program behaves as expected for a finite number of inputs, and those based on debugging where errors are detected and removed. The strength of testing and debugging is the ease of use while the drawback is that no guarantees that every error has been found and removed can be made. For regular applications, it is usually sufficient to demonstrate a reasonable level of reliability, and developers can save time up front by waiting for overlooked errors to reveal themselves.

For applications that perform critical tasks, for instance monitoring or controlling infrastructure, we need a higher level of reliability than what can be achieved by the usual methods. To achieve the highest level of reliability in software systems, we need the mathematical precision of formal verification. Formal verification of software systems is the act of proving their functional correctness with respect to a formal specification. While formal verification cannot help us make sure that "we are building the right system", i.e. that the system has been formally specified correctly, the problem of building the right system is not exclusive to formal verification; it is merely exposed through the act of formally specifying the system. Formal verification is incredibly powerful, but such power comes at a price. Firstly, most formal verification techniques require expert knowledge to apply. However, ongoing efforts strive to make formal verification more accessible. Secondly, formally verifying a complex soft-

ware system usually takes a long time. As a counterpoint, it can be argued that "getting it right the first time" can save overall development costs despite the initial cost of applying formal techniques.

A multi-agent system is a software system composed of multiple intelligent agents able to solve complex problems. By an agent we understand an entity that perceives its environment and autonomously takes actions to achieve its goals, possibly assisted by knowledge or the ability to learn. This definition is very open to interpretation and its applications extend beyond physical and virtual robots. A cognitive multi-agent system is a special kind of multi-agent system in which agents exhibit cognitive behavior. In recent decades, several research agendas have manifested focused around multi-agent systems and in particular agent-oriented programming based on mental models.

Multi-agent systems and agent-oriented programming are predominantly concerned with symbolic Artificial Intelligence (AI) approaches as dedicated agent programming languages and frameworks such as GOAL and JADE have emerged [6, 33, 36, 39]. In symbolic AI, models are explicitly represented, whereas in subsymbolic AI, models are implicitly represented. Machine learning is a subsymbolic AI approach which has gained a lot of traction, due to its recent successes, which has caused a resurgence of interest in AI. Furthermore, it has become easier for developers to apply machine learning techniques by means of easy-to-use frameworks such as TensorFlow and PyTorch [1, 60]. The success of machine learning has inspired grand visions for the future of AI, but Bordini, Seghrouchni, Hindriks, Logan and Ricci argue that machine learning cannot fully replace "programmed AI" [11]:

> It is claimed that, in the nascent "Cognitive Era", intelligent systems will be trained using machine learning techniques rather than programmed by software developers. A contrary point of view argues that machine learning has limitations, and, taken in isolation, cannot form the basis of autonomous systems capable of intelligent behaviour in complex environments.

Instead, they see an opportunity in integrating learning algorithms into the programmed AI which is considered more proficient at capturing high-level concepts. For instance in an agent where image recognition is performed by a learning algorithm while a programmed AI decides upon which actions to take.

Our focus is on multi-agent systems built from symbolic AI approaches, as their explicit programming more easily allow for the application of formal techniques to demonstrate their reliability. We know that demonstrating reliability of multi-agent systems is notoriously difficult. This is particularly true for cognitive

multi-agent systems, as they exhibit complex behavior patterns, which makes them very difficult to test, debug and formally verify. This thesis is motivated by the need for a new research agenda for demonstrating reliability of multi-agent systems, as argued by Dix, Logan and Winikoff [28]:

> The aim of this seminar was to bring together researchers from various scientific disciplines, such as software engineering of autonomous systems, software verification, and relevant subareas of AI, such as ethics and machine learning, to discuss the emerging topic of the reliability of (multi-)agent systems and autonomous systems in particular. The ultimate aim of the seminar was to establish a new research agenda for engineering reliable autonomous systems.

We approach this task by using a proof assistant, or interactive theorem prover, which is a software tool that assists the user in expressing and proving mathematical formulas in a formal language. Formal proofs are developed through interaction between human and computer. The role of the user is to guide the proof assistant in the proof search. The proof assistant offers tools to facilitate the guided search. Typically, this is paired with the use of automatic provers to discharge proof goals. A substantial part of the work in this thesis is developed in the proof assistant Isabelle/HOL [58] which is the higher-order logic instance of the generic proof assistant Isabelle [26]. Proofs in Isabelle are developed in the structured proof language Isar which is designed to be readable by both humans and computers. Isabelle incorporates several proof tools and the Sledgehammer tool can assist the proof development by searching for applicable proof methods to discharge the active proof goal. There are several other proof assistants, usually categorized by one of two major branches: those based on simple type theory such as HOL Light, HOL4 and Isabelle/HOL [32, 73], and those based on dependent type theory such as Lean, Coq and Agda [3, 7, 75].

This thesis is about the engineering of reliable multi-agent systems. We explore a new approach to ensuring reliability of agents in agent-oriented programming. We develop a complex multi-agent system in GOAL that competes at a research-focused competition for multi-agent systems. We then consider existing work on a verification framework for GOAL agents, and develop a method of transformation from specification of GOAL program code to GOAL programming logic. We further consider how a theorem proving approach, and the use of a proof assistant, can contribute to demonstrating reliability for multi-agent systems. We apply our own approach, and formalize said verification framework for GOAL agents in the Isabelle/HOL proof assistant. We apply the formalized verification framework to prove the correctness of an example agent. Finally, we reflect on the contributions, limitations and perspectives of our work from a desire to verify multi-agent systems such as the one we developed for the competition.

The introductory chapter is structured as follows. Section 1.1 introduces preliminary topics. Section 1.2 is a synopsis of the published articles. Section 1.3 outlines new work, exclusive to this thesis, which describes technical details of selected Isabelle proofs. Section 1.4 highlights selected other developments not included in this thesis. Section 1.5 discusses the contributions, limitations and perspectives of the work in this thesis. Section 1.6 is an overview of the chapters and Isabelle files.

1.1 Preliminaries

This section introduces the necessary preliminaries and is structured as follows.

Section 1.1.1 describes the concept of a multi-agent system. Section 1.1.2 describes agent-oriented programming. Section 1.1.3 describes the basic principles of the agent programming language GOAL. These topics are preliminaries to the work presented in Chapters 2 to 5.

Section 1.1.4 describes the formal verification of GOAL agents. This topic is a preliminary to the work presented in Chapters 3 to 5.

A central part of the work presented in this thesis is the development of an Isabelle formalization which is publicly available online [45]. See Section 1.6 for an overview of the chapters and Isabelle files.

1.1.1 Multi-Agent Systems

The study of agents and multi-agent systems has seen various, different approaches to the core concept of "multiple agents interacting in an environment". The environment is the context of agents, and the agents interact by taking actions. Both the actions agents can take and the rules of the environments vary between systems. In [70], Russell and Norvig seek to classify agents and environments to establish their recurring characteristics.

A multi-agent system is typically associated with the implementation of software to control agents but has seen various adaptations. The perhaps most intuitive adaptation is the deployment of physical agents in the real world, e.g. autonomous service robots such as vacuum cleaners. Even physical agents are not limited to robots in the classical sense: a sensor that reads and conveys data from the physical world to another agent can itself be modeled as an agent.

In the context of research and teaching, real world problems are often simulated by the use of virtual agents and environments. Other virtual adaptations include modeling of software systems as multi-agent systems where software components are seen as agents that operate in a system to solve a problem.

The study of logics for agents has close relations to the study of modal logics, in particular to epistemic logic for reasoning about knowledge of agents. Whereas epistemic logic is concerned with epistemic situations, dynamic epistemic logic seeks to model also the dynamics of knowledge by describing actions and their effects on the knowledge of agents. As such, dynamic epistemic logic can be used to reason both about information change and exchange of information between agents [27].

The many possible interpretations of the core agent concept has inspired numerous adaptations. In particular for cognitive agents, one such adaptation is to model agents as "logical inference engines". Such logical inference agents continuously expand their knowledge bases from known and learned inference rules and facts. Logical inference agents are subject to the well-known limitations and challenges of logic in general. In particular, there is the problem of logical omniscience where agents are constructed in such a way that they know all tautologies. As agents learn new facts, all logical consequences must be computed as well. Clearly, an infinite supply of tautologies does not bode well for computing all logical consequences when implementing such agents as software programs. Considering agents as resource bounded may be an effective way to deal with these problems in practice [74]. The focus on this thesis is on agent-oriented programming and the use of dedicated programming languages where agents only perform logical inference when it is required.

1.1.2 Agent-Oriented Programming

Agent-oriented software engineering is concerned with the development of software systems capable of exhibiting autonomy and rational behavior in dynamic and unpredictable environments. The agent-oriented approach to programming such systems typically employs special purpose agent programming languages that directly support concepts such as beliefs, goals, plans and communication between agents. In the current landscape, the predominant approach is for such programming languages to be based on the Belief-Desire-Intention (BDI) model [14]. The classical, early languages that have inspired later developments include AgentSpeak [66], which was directly inspired by the theoretical work on BDI logics [67, 68], and 3APL where agents are capable of self-modifying their plans [35].

The BDI model is used to model the behavior of a cognitive agent. In this model, agent behavior is specified in terms of beliefs, goals, intentions and plans. Beliefs represent the agent's static knowledge and gathered information. The representation of goals takes one of three forms:

1. An achievement goal describes a state the agent is trying to achieve.
2. A maintenance goal describes a state the agent is trying to maintain.
3. A procedural goal describes a desired course of action.

Plans describe how an agent responds to certain events and are typically programmed by the developer. The head of a plan is a rule which describes when the plan should be considered. The body of a plan may consist of basic actions, test conditions and subgoals. Together with its beliefs and goals, the agent's plans model the agent program. Intentions represent the agent's commitment to future courses of action, usually implemented as a stack of plans. At run-time, the interpreter is responsible for updating the agent's beliefs and goals based on messages from other agents and sensory information, managing the intentions, choosing plans and executing actions.

There are several popular programming languages based on the BDI model. GOAL is an agent programming language which is based on a formal semantics and uses a declarative notion of goals (achievement and maintenance goals) [36, 39]. There is a family of languages based on AgentSpeak, such as Jason [10] and ASTRA [20], which both extend the core functionality with additional features.

Standard frameworks such as the Environment Interface Standard (EIS) facilitate a seamless development of connection protocols for agents that connect to external environments including games and robots [4, 5]. EIS lessens the burden of writing so-called "glue code" that connects agents to their environments by providing a standard for the agents to send actions and receive so-called "percepts" (a collective term for different types of information agents receive from the environment).

In this thesis, we focus on the development of agents using the agent programming language GOAL. Because it is based on a formal semantics, it provides a unique opportunity for formal verification of agents.

1.1.3 Programming Agents in GOAL

GOAL is a dedicated agent programming language which supports testing, debugging and programming in the Eclipse IDE and connects to EIS-enabled environments [25].

A multi-agent system in GOAL is specified by a configuration file which describes launch and termination conditions for agents and optionally an environment to connect to. Agents are specified through a number of modules. Typically, an initialization module instantiates the beliefs and goals. Several event modules may trigger based on changes to the environment and update the agent's state accordingly. Finally, a main module selects which action to perform based on decision rules. An action specification specifies the actions including pre- and postconditions. This layer is usually minimal for systems that connect to an environment. At run-time, the execution of agent programs is performed in reasoning cycles of event processing, decision-making and action performing.

The cognitive model of GOAL includes the familiar BDI concepts of beliefs and declarative goals ("goals-to-be"). GOAL does not explicitly include a concept of intentions and thus deviates from the classic BDI model. In [34], it is argued that modules can be seen as a form of intentions, and the deviation allows for a simpler reasoning cycle. The plans of GOAL are its decision-rules for actions. These are simpler than plans in the classic formulation of the BDI model.

The GOAL programming language and programming logic, or agent logic, are related by means of a formal semantics [22]. The programming logic allows for verification of agents and, because a relation exists between the two, we can be certain that statements proven in the logic are in fact properties of agents.

1.1.4 Formal Verification of GOAL Agents

The two major branches of formal verification of GOAL agents, and cognitive agents in general, are those based on either model checking or theorem proving using deductive reasoning. Most of the work on verification of multi-agent systems has been with limited logical languages such as by Alechina, Dastani, Khan, Logan and Meyer [2], and Shapiro, Lespérance and Levesque in [71].

Model checking is the act of validating properties in all states of a system. Usually these properties are expressed and validated using a temporal logic. When research in multi-agent systems gained traction, adopting model checking was an obvious choice for verification of agent systems. Model checking had already

proven effective for verifying traditional software systems [18]. To this date, model checking remains the state-of-the-art for verification of cognitive agents [17, 55]. A framework is proposed for applying model checking to BDI-based agent programming languages by Dennis and Fisher in [24], and in particular for GOAL, Jongmans, Hindriks and Riemsdijk apply model checking directly on top of the program interpreter in [52], but the framework is applicable to other agent programming languages as well.

The other major branch of formal verification is theorem proving, which is the act of validating properties by deductive reasoning. Whereas model checking explores all possible states of the system, theorem proving approaches try to be independent from concrete states. We call this "automated theorem proving" when proving (or disproving) theorems is performed automatically by a computer. Interactive theorem proving in a proof assistant combines "manual" and automated theorem proving: the interaction between user and proof assistant guides the proof search where the proof goals are eventually discharged, often by invoking an automated prover on a subgoal. The maturity of state-of-the-art proof assistants such as Isabelle/HOL motivates us to explore a theorem proving approach for verification of multi-agent systems.

The GOAL language is based on a formal semantics which facilitates the use of formal techniques for verification that can be formalized in a proof assistant. In particular, the work by de Boer, Hindriks, van der Hoek and Meyer in [22] describes a "pen and paper" formulation of the formal semantics of GOAL and its programming logic. The formulation of a programming logic enables verification of agents and, uniquely for GOAL, the programming logic is provably related to the programming language. By formalizing this work in the proof assistant Isabelle/HOL, one benefit is that we gain confidence that the framework works as intended as everything is checked by the proof assistant. A second benefit is that a programmed version of the verification method can be used as a "software tool" for single-stepping proofs. The strongest result is that these two benefits exist side-by-side: we can verify agents using a tool that has been checked by the proof assistant.

1.2 Synopsis of Published Articles

Chapters 2 to 5 contribute towards more reliable engineering of multi-agent systems using GOAL. Chapter 2 in particular contributes towards research in engineering multi-agent systems by developing GOAL agents for competing in the 14th Annual Multi-Agent Programming Contest (MAPC 2019). The contest scenario involves agents moving about in a dynamic environment with limited

vision where they must cooperate to build complex patterns from blocks. The programming of agents requires creativity as the scenario is designed in such a way that there is no obvious solution.

Chapters 3 to 5 contribute to the demonstration of reliability of agents in general, and in particular their contributions revolve around a technique for formal verification of GOAL agents. Chapter 3 stands out by contributing to the development of a method of transformation from actual GOAL agent program code to a logic for agents.

Chapters 4 and 5 contribute with a formalization of the aforementioned formal technique for verification of GOAL agents in the proof assistant Isabelle/HOL. Along with my paper [44], they form in cohesion a chronological development report on the formalization, but they each focus on different aspects of it. Chapter 4 contributes by exploring a theorem proving approach, assisted by a proof assistant, as a formal technique for demonstrating reliability of agents. Chapter 5 contributes with a formalization of a verification framework for GOAL agents. Furthermore, it contributes by applying the formalized verification framework to verify the correctness of an example agent program.

Development of Distributed Intelligence for the Multi-Agent Programming Contest

Chapter 2 is the published article "GOAL-DTU: Development of Distributed Intelligence for the Multi-Agent Programming Contest" by myself and Villadsen [49]. The paper is about the development of a multi-agent system for the MAPC 2019 where we placed 3rd [59]. We used GOAL to program our agents while the other teams used either JaCaMo, Jason or Java. The scenario "Agents Assemble" concerns multiple agents in a team moving about a dynamic grid map with limited vision. Agents are placed randomly across the map without knowledge about the position of the other agents on the team. Blocks are collected across the map to complete tasks. Completing a task awards points and requires that certain block types are aligned in a specific pattern. Tasks can only be submitted in special goal cells which must be located on the map. The winning team is the one with the most points at the end of the simulation.

I was the sole developer, and understanding how to program multi-agent systems using GOAL was a central part of my studies. The system was built from scratch. As there was a limited amount of time to develop the system, we were forced to think in creative ways in terms of finding effective solutions while minimizing the development time. For instance, our agents employ a simple heuristic to move towards target positions, and only recently visited positions

are stored as knowledge in an effort to steer our agents away from repeating the same (nonconstructive) movement patterns. In contrast, other participants opted to slowly build up a representation of the map over time and use this to plan routes. In comparison, our findings show that our agents were better at reacting to changes, as we did not commit to any plans or routes, while agents that used route planning were more vulnerable to changes but in general moved along more optimal paths.

The 15th Annual Multi-Agent Programming Contest (MAPC 2020/21) extended the scenario from the previous year. We competed again in 2020/21 with an agent system built from the code base we used to compete in the MAPC 2019 [50]. With our supervision, students working on their bachelor theses improved upon the system and improved the performance of the agents. We placed on a shared 2nd out of the five participating teams. A 16th edition of the MAPC in 2022 have been announced, with a revised version of the previous scenario, and current plans are for students to continue development on our system to address the new challenges in the revised scenario.

The contribution of the paper is to describe the design of our agents. Furthermore, through evaluation of the matches that were played against opposing teams, we are able to provide practical insights into which design choices worked well and which did not. We finally reflect on the limitations and perspectives of further development in relation to the design choices.

Towards Verifying a Blocks World for Teams GOAL Agent

Chapter 3 is the published article "Towards Verifying a Blocks World for Teams GOAL Agent" by myself [43]. The paper is concerned with the use of a verification framework by de Boer, Hindriks, van der Hoek and Meyer in [22]. The primary contribution is that we describe a method for transforming the program code of a single GOAL agent into the programming logic, such that we can apply the verification framework to verify its correctness. We further explain how certain assumptions must be made about the code structure, the agent and the environment for the transformation to be possible.

The other contribution of the paper is to develop a proof-of-concept by applying the transformation on a small example GOAL program, and we sketch how to prove its correctness.

Lastly, we describe some of the work ahead in order to automate the transformation entirely, and to achieve the vision of being able to transform and prove complex multi-agent programs using the verification framework.

On Using Theorem Proving for Cognitive Agent-Oriented Programming

Chapter 4 is the published article "On Using Theorem Proving for Cognitive Agent-Oriented Programming" by myself, Hindriks and Villadsen [47]. The paper suggests an approach to formal verification of agents using interactive theorem proving. That is, a theorem proving approach that is assisted by a proof assistant. We focus on the GOAL language as it has a number of advantages. Firstly, the GOAL language is based on a formal semantics for which it can be proved that a relation exists between its programming language and logic. The programming logic enables us to verify agents using the aforementioned verification framework by de Boer, Hindriks, van der Hoek and Meyer in [22]. The provable relation implies that in doing so, we in fact verify a GOAL agent program. Secondly, the formal semantics of GOAL is relatively simple which makes it easier to formalize in a proof assistant.

We also reflect on why theorem proving has not been explored further for verification of agents. In particular, we observe that theorem proving and proof assistants have successfully verified various software and hardware systems [69].

Lastly, we describe the early efforts of our work on formalizing the verification framework in Isabelle/HOL.

Machine-Checked Verification of Cognitive Agents

Chapter 5 is the published article "Machine-Checked Verification of Cognitive Agents" by myself [46]. This paper is the most recent entry in the sequence of papers about our development of an Isabelle formalization of the verification framework by de Boer, Hindriks, van der Hoek and Meyer in [22]. The formalization covers every part of the original paper required to verify agents. In particular, the focus is on verifying correctness properties of agents.

In the paper, we go through the formalization from start to finish. We begin with the low-level semantics of GOAL and end with a temporal logic which enables us to prove key temporal properties of agents in the programming logic.

Finally, we show how to verify agents in Isabelle/HOL. As an example, we specify a simple agent in the programming logic. This agent is based on the GOAL agent program code considered in Chapter 3, but it has been simplified to make the proof simpler. The main result is a formalization of the verification framework and a correctness proof of the example agent.

1.3 GOAL in Isabelle/HOL: Selected Proofs

Chapter 6 is unpublished work written by myself, exclusively for this thesis. The work extends Chapter 5 with detailed descriptions of two selected Isabelle proofs. The chapter provides insight into some of the technical work in Isabelle that leads to the results presented in Chapter 5. The chapter does not present new results and may be considered optional.

The first proof is an integral theorem of the verification method which states that safety properties concerning a temporal operator *unless* are equivalent to a finite set of Hoare triples. In other words, it shows that we can prove certain safety properties using the programming logic for GOAL. The second proof shows the relation distinction between Hoare triples for basic actions and for conditional actions.

1.4 Selected Other Developments

This section highlights selected other developments during or closely related to my PhD studies.

The paper "Teaching a Formalized Logical Calculus" by From, Schlichtkrull, myself and Villadsen [30] describes how to formalize a logical calculus in Isabelle/HOL from a perspective of teaching it to students. The topic relates to Chapters 4 and 5 as it applies some of the same principles to formalize the logic. The paper is an extension of a paper published prior to the PhD studies by Schlichtkrull, myself and Villadsen [81].

The techniques used for formalizing logic also draw much inspiration from the paper "Programming and Verifying a Declarative First-Order Prover in Isabelle/HOL" by myself, Larsen, Schlichtkrull and Villadsen, also published prior to the PhD studies, which is based directly on the work of my master's thesis "First-Order Logic According to Harrison" [48]. The paper "Interactive Theorem Proving for Logic and Information" by Villadsen, From, myself and Schlichtkrull [80] presents a formalization of epistemic logic in Isabelle/HOL.

Chapters 3 to 5 are all based on the work presented in "Formal Verification of a Cognitive Agent Using Theorem Proving" by myself [42].

1.5 Contributions, Limitations and Perspectives

The work in this thesis contributes towards advancing formal verification techniques for multi-agent systems:

1. **Competition multi-agent system**. The system developed in Chapter 2 contributes towards the aim of the MAPC to stimulate research in the area of multi-agent system development and programming. Furthermore, it serves as a benchmark for the complexity of systems we want to be able to verify. We provide practical insights into the design choices of our system.
2. **Transformation method**. The transformation method described in Chapter 3 lays the foundation for a mechanical transformation from GOAL program code to agent logic. When the GOAL program is specified in agent logic, it enables verification of the agent.
3. **Formalization in Isabelle/HOL**. The Isabelle/HOL formalization of the verification framework developed in Chapters 4 and 5 proves soundness with respect to the formal semantics of GOAL. It further enables us to develop proofs in the agent logic in Isabelle/HOL. The formalization lays a foundation for further work and experimentation. To our knowledge, it is the first formalization of an agent verification technique in a proof assistant.
4. **Example proof in Isabelle/HOL**. The example proof in Chapter 5 verifies the correctness of a small example agent in Isabelle/HOL.

This section discusses limitations and perspectives of our contributions in regards to future challenges. The multi-agent system developed for the MAPC 2019 serves as a benchmark for systems we want to be able to verify. We identify the following key challenges and discuss them in the coming sections.

1. **Multiple agents and communication**. The verification framework only applies to single agents. For systems consisting of multiple agents, usually their behavior is cooperative and a single agent cannot be considered in isolation.
2. **Programming with variables**. GOAL is heavily influenced by logic programming and allows for programming with variables. The verification framework is based on propositional logic which causes both theoretical and practical limitations.

3. **Environments and non-determinism**. The verification framework assumes that agents are the only ones capable of changing their mental state. Environments must be integrated into the logic of our agents. Furthermore, the uncertainty of such environments is not well supported.

4. **Deadlines and durative actions**. The programming language for GOAL provides no inherent way for agents to reason about time and durative actions such that we can guarantee that deadlines are met. We reflect on whether encoding time-based events in propositional logic is an appropriate abstraction.

5. **Advanced programming concepts**. The verification framework is based on a simplified version of the GOAL programming language. More advanced programming concepts such as structuring into modules are left out.

6. **Proof automation**. Developing proofs in the verification logic requires single-stepping proofs. Automating parts of the proof development would make it less tedious and more accessible.

In the following, we reflect on how to overcome each of the identified challenges.

1.5.1 Multiple Agents and Communication

In order to model and verify systems consisting of multiple agents that communicate, we take inspiration from the work by Bulling and Hindriks in [15, 16] which is an extension of the work on a verification framework for GOAL agents by de Boer, Hindriks, van der Hoek and Meyer in [22]. Here, we describe and reflect on their work in relation to our own.

The state of a GOAL agent program is extended to be a global state which is a tuple containing the mental state of each agent. The mental state of each individual agent is extended to contain a partial model of the mental state of each other agent. The model is partial because the model is only updated based on the communicative actions of other agents. The formal semantics is extended such that the modal operators also allow for beliefs about the mental state of other agents.

Bullings and Hindriks' approach assumes that agents operate by the cooperative principle, namely that they only tell (what they believe to be) truths. The effect of communicative actions are dealt with by the message receiver, such that upon receiving a message Φ, the receiver comes to believe that the sender believes Φ.

The authors argue that this decision is made from an engineering perspective since an agent cannot access nor modify the mental state of another agent.

Messages use one of three indicators that correspond to naturally occurring sentence types: • for declarative, ? for interrogative, and ! for imperative sentences. Messages are thus of the form $i\Phi$ where i is one of the three indicators. If an agent A sends a message, the receiving agent B's response depends on the message type:

- $\bullet\Phi$: Φ suggests that A believes Φ and it is added to B's internal representation of the beliefs of A while entailed subgoals are removed.
- $?\Phi$: Φ suggests that A does not believe Φ when it asks a question Φ and it is removed from B's internal representation of the beliefs of A.
- $!\Phi$: Φ suggests that A wants to achieve Φ; it is added to B's internal representation of the goals of A (if it is not a tautology).

Bulling and Hindriks then define an extended verification logic and prove that it can be embedded into the verification logic. They acknowledge that further work is required. They provide no examples so it remains to be seen how this plays out in practice.

If we compare the approach above to actual programming of agents in GOAL, we notably see two differences. Firstly, programmed GOAL agents do not inherently maintain mental models of other agents. Secondly, whereas the sentence types (moods of messages) above enforce a certain behavior, the moods of messages are merely suggestive in the programming language, and developers of agents are free to react to messages as they see fit.

The suggested approach contributes to advance the capabilities of the verification framework by laying a foundation for the verification framework to work for multiple communicating agents. Meanwhile, the gap between actual programming of GOAL agents and their specification in the framework seems to widen as we require that partial mental states of other agents are maintained. We speculate that this gap could be bridged by specifying communicative actions more like in the programming language. However, it is unclear what impact such modifications would have on the verification logic.

1.5.2 Programming with Variables

In the following we reflect on extending the logic of the verification framework to support programming with variables.

In programming terms, the mental state of agents consists of beliefs, (fully instantiated) goals and static knowledge. The verification framework incorporates static knowledge as beliefs because the distinction provides no theoretical benefits. We will assume that Prolog is used to represent knowledge.

Because goals are ground they contain no free variables and can be represented by predicate logic with no variables or quantifiers. Beliefs are formulas which may contain free variables that are implicitly considered as universally quantified. This suggests that mental states should be two sets of different types of formulas because only beliefs can have variables.

For queries to inspect the mental state of agents, we assume an implicit existential quantification. For instance, the query $\mathsf{B}(Px)$ succeeds if $\Sigma \models \exists x.\ P(x)$ for a belief base Σ. Importantly, note that GOAL and Prolog use negation-as-failure and the "closed world assumption", i.e. absence of $P(a)$ in the belief or goal base implies $\neg P(a)$.

Because a mental state transformer is an abstraction over actions and event handling, an enriched language for beliefs should be straightforward to formalize. However, for the specification of Hoare triples, we need to consider the implicit meaning of free variables. The intuitive approach seems to be that variables are bound across the entire Hoare triple such that $\{\mathsf{B}(Px)\}\ a\ \{\neg\mathsf{B}(Px)\}$ states that action a negates all occurrences of Px. We need to carefully consider if the Hoare system needs to be modified or extended. The temporal logic should remain unchanged.

We acknowledge that the reflections above may have overlooked potential challenges and pitfalls. Our reflections can used as inspiration for future work, but further studies are required.

1.5.3 Environments and Non-Determinism

The environments agents operate in play a central part in their behavior in multi-agent systems. Physical agents that operate in the real world rely on sensors to observe the state of their surroundings whereas purely software-based agents connect to virtual environment servers. For virtual servers, we may have partial

or full knowledge of the environment. In the physical world, the surroundings may be constrained such that we can apply basic physical principles to predict outcomes. However, both approaches are prone to error as unexpected events may occur. In a software environment, there may be an unstable connection to the virtual server, and in the physical world, for instance a factory worker agent may encounter problems with the assembly line. In both cases, we should attempt to foresee such events and build agents with robustness and error-correction in mind.

In the context of multi-agent system verification, how to build frameworks that incorporate environments has not been addressed properly. Usually, the state of affairs is incorporated into the mental models of agents and their action theories. Theoretically, it poses no immediate problems to incorporate knowledge of the world into the mental models of agents. However, there may be conceptual benefits from separating the state of the environment, and the rules that govern it, from the state of each individual agent. In particular when we consider multiple agents, maintaining an updated state for each agent seems more cumbersome than maintaining a single state of the environment. While we do not have any proposals for the first steps towards such conceptual extensions of our verification framework in particular, it is an interesting topic for future studies.

Non-determinism in multi-agent systems often appears in environments of agents in the form of actions that unexpectedly fail. Uncertainty and probability in logic has already been studied. The first appearance of the term "probabilistic logic" is in [57] by Nilsson, where truth values are probabilities, which reduces to ordinary logic when truth values are either 0 or 1. Many-valued logics have been researched extensively throughout the years [76]. Fuzzy logic, as proposed by Zadeh [86], is a many-valued logic where truth values are real numbers between 0 and 1. As opposed to probabilistic logic, "fuzziness" insinuates a vagueness of the propositions themselves. The well-known 3-valued logic, with the additional truth value *maybe*, avoids the intricacies of real number calculations as all probabilities between 0 and 1 are represented by a single value.

For the purpose of GOAL agents, our main interest lies in addressing certainty of outcomes of actions and not in addressing certainty of beliefs and knowledge. How to approach certainty of outcomes depends on which properties we want to show. For absolute reliability, all possible outcomes need to be considered and probabilistic values are thus reduced to either true or false, which suggests that no extensions are needed. For reliability as a probability that a property holds, we need a stronger logic for agents, and it should be explored if inspiration can be taken from probabilistic logic. For properties stated in a 3-valued logic, using the truth value *maybe*, we also need a stronger logic for agents, but such an extension would be less extensive, while the drawback is that showing that a property *maybe* holds may not be satisfactory.

1.5.4 Deadlines and Durative Actions

The success or failure of agents may be determined by whether they are able to complete tasks within certain deadlines. In such cases, verifying an agent means proving that deadlines are never exceeded. If non-determinism is involved, it may instead be a case of proving that the probability of failure is below a certain threshold.

The failure to meet a deadline can be modeled as an observable event that occurs in the environment. In the temporal logic constructed on top of GOAL, such a property may be expressed as a safety property which can be reduced to proving a finite number of Hoare triples. The question is then how to express the occurrence of a failure event. Due to the limited logic language of agents, they are not inherently equipped to reason about (linear) time. For a finite propositional language, a discrete number of time units can be represented in cumbersome ways, but a stronger logical language for agents is needed for a more general approach.

Another related issue is durative actions. If we consider the examples used in Chapters 3 and 5, we assume that actions occur instantly (their effects are in place in the following time step). Both examples are inspired by agents built towards completing tasks in the Blocks Worlds for Teams (BW4T) environment [51] in which actions are durative. In particular for actions that involve the agents moving about the map, their duration depend on the distance traveled.

When we know that the action execution completes within a number of steps, an encoding using a number of intermediate states and actions could potentially solve the case. It is more problematic when the execution time depends on environmental variables, such as the distance to travel. In particular when such variables are not observable. For instance, a virtual environment may perform its own computations to execute a requested move action. The details of such computations will be a black box to the agent. Firstly, it seems that a stronger logic language for the agents would be helpful or even necessary. Secondly, we clearly need some knowledge about the duration of the action to be able to verify that the agent will complete the task within the deadline. The question remains if the logic of the verification framework is appropriate for dealing with such time constraints, or if it is necessary to incorporate a concept of time explicitly.

1.5.5 Advanced Programming Concepts

The verification framework is based on a simplified programming framework for GOAL that leaves out some of the more advanced programming concepts. Some of the concepts left out have already been discussed in previous sections:

- Programming with variables is discussed in Section 1.5.2.
- Communication between agents is discussed in Section 1.5.1.
- Environments are discussed in Section 1.5.3.

We reflect in the following on how to integrate some of those programming concepts into the verification framework.

Communication channels. As opposed to sending messages to one specific agent, a common optimization method is for agents to subscribe to channels. Messages sent to channels provide an optimized and intuitive way of sending messages to a designated group of agents. The work we considered in Section 1.5.1 should be easily extendable to include a concept of channels, although it is not clear if doing so would provide theoretical benefits. If security is not a concern, this could easily be simulated by sending the message to all agents and naming the recipients as part of the message.

Structuring into modules. GOAL allows for structuring of code into modules. The programming flow allows switching between modules and it is used to both structure code and create focus for agents. By focus we mean that the agent "switches" to different modules that are programmed to deal with particular goals. Outside of making the programming framework as close to the programming language as possible, there is little benefit in integrating the concept of modules into the verification framework.

Settings of agents. Agents in GOAL may be instantiated with a plethora of different settings such as launch policies and rule evaluation orders (deciding which action is selected if multiple are applicable). In the verification framework, we assume any possible scheduling of actions, but only consider those that are enabled. In the case that multiple actions are enabled, the proof branches, as we consider both outcomes possible. Launch policies are not considered as we only deal with runs of single agents. In choosing which of these settings to possibly

integrate into the verification framework, we need to carefully think about the potential overhead and theoretical challenges compared to the benefits.

Proof Automation

The example proof presented in Section 5.7 is developed by single-stepping proof rules of the Hoare system. Even for our small example, it requires a lot of work to develop the proofs of the Hoare triples. The automation of Isabelle/HOL can easily discharge subgoals when they concern the semantics of the propositional logic. However, the inductive proof system for proving Hoare triples is too complex even when invoking the Sledgehammer tool.

The "Eisbach" collection of tools allows for defining new meta-level proof methods in Isabelle (the user manual can be found on the "Documentation" page on the Isabelle web page [26]). The methods are written in Isar syntax and can be abstracted over terms, facts and other proof methods. Additional functionality such as pattern matching is provided as well.

In the following, we reflect on the proof rules of our Hoare system in relation to writing meta level proof methods to assist in the development of proofs of Hoare triples. The proof rules are most naturally applied backwards, starting from the Hoare triple to prove. The Hoare system can be found in the *Gvf_Agent_Specification* theory in the formalization [45] and is informally described in Section 5.5.2. It corresponds directly to the proof system presented in the seminal paper by de Boer, Hindriks, van der Hoek and Meyer in [22].

Axioms. The axioms of the Hoare system are for:

- import of agent specific axioms,
- showing persistence of goals,
- showing infeasibility of actions,
- showing properties of actions for manipulation of the goal base.

For all of the axioms, pattern matching can easily determine if they apply, while some axioms also require checking a simple condition. As such, automating the application of such rules should be simple.

Conditional actions. The rule for conditional actions states that a Hoare triple for a conditional action, which is an action that is only enabled when a condition holds, can be proved by a Hoare triple for a basic action, which is an action with no condition, by proving that the negation of the condition in conjunction with the precondition always leads to the truth of the postcondition. In general, determining the truth of such an implication is simple for Isabelle. Furthermore, it is always progressive to apply the rule which means that pattern matching can be used to automate the rule.

Structural rules. The difficulty of providing a fully automated proof tactic lies in the structural rules. The rule *rImp* allows for strengthening of the precondition and weakening of the postcondition. It is not always obvious how this rule should be applied. Examining its uses in the proof of our example agent reveals the following uses:

1. Removing unnecessary conjuncts in preconditions.
2. Removing unnecessary disjuncts in postconditions.
3. Duplicating conjuncts in preconditionss.
4. Rearranging the order of conjunctions and disjunctions (necessary because of the strictness of the other structural rules).

In general, there are infinitely many possible applications of the *rImp* rule. The different use cases suggest that a breadth-first search could ensure that we eventually find a proof if it exists. Whether this is indeed the case requires further studies.

The rule *rCon* splits the proof of a Hoare triple into two when both the pre- and postcondition are conjunctions. The rule *rDis* splits the proof of a Hoare triple into two when the precondition is a disjunction. Both rules are based on pattern matching which means that they apply differently based on the order and nesting of elements, i.e. $p_1 \wedge p_2$ is different from $p_2 \wedge p_1$ and $p_1 \wedge (p_2 \wedge p_3)$ is different from $(p_1 \wedge p_2) \wedge p_3$. A naive approach to an automated tactic is to try all different combinations (which can be generated by *rImp*). However, a problem occurs if the tactic for *rImp* does not guarantee termination (which a full search would not). Both *rCon* and *rDis* have a finite number of applications when considering the different combinations. This suggests to either make sure that a tactic for *rImp* eventually gives up on seemingly dead ends, or alternates between the different options of *rCon* and *rDis*. We have not looked into how or if this behavior can be achieved using Eisbach.

1.5.6 Concluding Remarks

In this section, we identified the key challenges to address in future work towards being able to verify multi-agent systems such as the one we developed for the MAPC 2019. We reflected on the challenges and discussed how to address them.

- For verifying multiple communicating agents, we reflected on the work of Bulling and Hindriks in [15, 16].
- For programming with variables, we considered an approach using free variables with implicit quantification.
- For incorporating environments and dealing with non-determinism, we reflected on the conceptual problem of environments in theoretical frameworks, and we considered the use of non-determinism in logic and verification.
- For reasoning about deadlines and durative actions, we reflected on the need for agents to incorporate a notion of time.
- We highlighted some of the advanced programming concepts of GOAL, and reflected on each of them individually.
- Lastly, we reflected on how to automate proofs of Hoare triples, and discussed the challenges of the different rules in the Hoare system.

Our reflections lay a foundation for exploration of extensions to the verification framework that address these challenges.

1.6 Overview of Chapters and Isabelle Files

Chapters 1 and 6 are unpublished works written by myself, exclusively for this thesis.

Chapters 2 to 5 each corresponds to a published article with only formatting changes (listed in Appendix C).

- Chapter 2: *Alexander Birch Jensen and Jørgen Villadsen. GOAL-DTU: Development of Distributed Intelligence for the Multi-Agent Programming Contest. In Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, and Tabajara Krausburg, editors, The Multi-Agent Programming Contest 2019, Lecture Notes in Computer Science, pages 79–105. Springer, 2020* [49].

- Chapter 3: *Alexander Birch Jensen. Towards Verifying a Blocks World for Teams GOAL Agent. In Ana Paula Rocha, Luc Steels, and Jaap van den Herik, editors, Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021), volume 1, pages 337–344. Science and Technology Publishing, 2021* [43].
- Chapter 4: *Alexander Birch Jensen, Koen V. Hindriks, and Jørgen Villadsen. On Using Theorem Proving for Cognitive Agent-Oriented Programming. In Ana Paula Rocha, Luc Steels, and Jaap van den Herik, editors, Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021), volume 1, pages 446–453. Science and Technology Publishing, 2021* [47].
- Chapter 5: *Alexander Birch Jensen. Machine-Checked Verification of Cognitive Agents. In Ana Paula Rocha, Luc Steels, and Jaap van den Herik, editors, Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022), volume 1, pages 245–256. Science and Technology Publishing, 2022* [46].

The Isabelle source code has been tested in Isabelle2021-1 and is available online at my personal DTU web page:

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

The source code consists of the following files.

File name (*.thy)	**Short description**	**Lines of code**
Gvf_Logic	Propositional logic	268
Gvf_Mental_States	Mental states of agents	236
Gvf_Actions	Actions of agents	552
Gvf_Hoare_Logic	Hoare logic for agents	191
Gvf_Agent_Specification	Specification of agents	795
Gvf_Temporal_Logic	Temporal logic for agents	318
Gvf_Example_BW4T	Example correctness proof	1093
	Total:	3463

This chapter is a published article. See Section 1.6 for details.

Chapter 2

GOAL-DTU: Development of Distributed Intelligence for the Multi-Agent Programming Contest

We provide a brief description of the GOAL-DTU system for the Multi-Agent Programming Contest, including the overall strategy and how the system is designed to apply this strategy. Our agents are implemented using the GOAL programming language. We evaluate the performance of our agents for the contest, and finally also discuss how to improve the system based on analysis of its strengths and weaknesses.

2.1 Introduction

In fall 2019 we participated as the GOAL-DTU team in the annual Multi-Agent Programming Contest (MAPC). We are using the GOAL agent programming language [36, 37, 39] and we are affiliated with the Technical University of Denmark (DTU). We participated in the contest in 2009 and 2010 as the Jason-DTU team [13, 77], in 2011 and 2012 as the Python-DTU team [29, 83], in 2013 and 2014 as the GOAL-DTU team [82], in 2015/2016 as the Python-DTU team [79] and in 2017 and 2018 as the Jason-DTU team [78].

In 2019 we had the new *Agents Assemble* scenario. The paper is organized as follows:

- Section 2.2 describes agent programming using the GOAL language.
- Section 2.3 covers the overall strategy of our agents.
- Section 2.4 describes the knowledge our agents acquire from the environment.
- Section 2.5 describes how our agents communicate.
- Section 2.6 describes the movement of our agents.
- Section 2.7 covers how our agents complete selected tasks.
- Section 2.9 discusses improvements to the system.
- Section 2.10 is our conclusion.

2.2 Agent Programming in GOAL

This section introduces the basic concepts of the GOAL agent programming language that are relevant to the implementation of our system.

Agents in GOAL are to be understood as self-controlled independent entities. Each agent interacts with the environment and communicates with other agents. Percepts and messages are treated as events that can be processed. This event processing then feeds into the knowledge, beliefs and goals (the three components that comprise an agent's cognitive state).

The programming philosophy behind GOAL is quite different when compared to other popular agent programming languages. Beyond the cognitive state, the core abilities of an agent are the just mentioned event processing capability, the ability to represent knowledge and reason about it, and finally, rule-based decision-making which allows an agent to select an action based on its current cognitive state.

2.2.1 The GOAL Execution Loop

GOAL features a simple execution loop of each agent. Beyond an initialization module that can process the initial state of the environment, and set up the

initial cognitive state, GOAL follows the execution loop below:

1. **Check new events**: If there are no new events, the next step is skipped.
2. **Process events**: The event module processes new events. Recall that these events are either percepts from the environment or messages from other agents. It is the purpose of this module to update the cognitive state of the agent before selecting the next action
3. **Action selection**: The main module defines the rules for decision-making. Based on the rules, the first valid action is selected. Note that several actions may be applicable based on the rules in the main module. GOAL allows for other strategies, but it is essential to our implementation that the first action is always the one that is selected.
4. **Perform action**: The selected action is sent to the environment (and communication actions are executed internally).

Technically speaking, there is also a final step that applies the post-conditions (effects) for the selected action from our action specification. However, it is not relevant for us since we rely solely on the event module to perceive the effects of actions.

2.2.2 Action Selection

GOAL advocates that agents are individual entities that reason about their environment. They react to changes in their environment rather than executing predetermined plans. For example, an agent may devise a plan for a goal to be achieved. At some intermediate step in the plan, the next step may no longer be applicable due to (unforeseen) changes in the environment. GOAL tries to avoid the complexity of rebuilding "broken plans" by advocating a reactive model. We should consider how the agent can select appropriate actions based on the current state of affairs. However, note that it is still possible for programmers to represent plans via the cognitive state of agents using Prolog, but it is not facilitated explicitly by the language. The reactive approach is not flawless either: it can be difficult for programmers to come up with logical rules that produce the desired behavior, but by overcoming this challenge, we often have a more flexible agent.

2.3 Strategy

In our current system, we have a universal agent type. By a universal agent type we mean that all agents share the same logic. While it is possible to have different kinds of agents in GOAL, i.e. via modules, it is not something we currently utilize. Some advantages of a universal agent type are that it is faster to implement, every agent is seamlessly capable of everything and we need not worry about when to switch the agent's type. The main disadvantage is that the code base becomes convoluted as development progresses due to growing array of logical rules for selecting the appropriate action.

During action selection, the agents apply heuristic measures to determine movement directions. We describe the different variants of heuristic functions in Section 2.6. It should also be noted that we currently do not perform any *clear* actions. We did not manage to implement use of the action in a meaningful way for the contest.

The following priority list describes the decision-making process of our agents (with some simplifications) where the first applicable rule determines the action to be selected in a given state:

- If the agent is assigned to a task:
  - Detach any attached blocks not needed for the task. The agent will only detach blocks if it considers it non-obstructive to future movement. If not, it will continue to move (using the detach movement heuristic) until it considers it safe to detach.
  - Rotate the block into the position dictated by the task plan. If rotation is blocked, move until rotation is possible (using the exploration movement heuristic).
  - If the agent observes part of the pattern to be handed in, or if the agent is assigned to submit the task (the submitting agent), and is on a goal, wait for other agents (by performing the *skip* action).
  - If the agent observes the entire pattern, connect with other blocks/agents as described by the task plan.
  - If all blocks in the pattern are connected, the assigned agent submits the task.
  - If the agent finds the submitting agent (waiting in a goal area), move until the attachment(s) form the (partial) pattern (using the task pattern movement heuristic).

  - If the agent is the submitting agent, move towards a goal area (using the go to movement heuristics).
  - Else, move towards the position of the submitting agent (using the go to movement heuristics).
  - If a goal area is known, move towards it (using the go to movement heuristics).
  - Move into the most promising direction (based on the exploration movement heuristic).

- If the agent is not assigned to a task:
  - If a block or dispenser is in vision, and the agent does not have four blocks:
    * Rotate such that a free attachment spot is facing the direction of the block/dispenser. If rotation is blocked, move (using the exploration movement heuristic).
    * If the agent is next to a block, attach it to the agent.
    * If the agent is next to a dispenser, request a block.
    * If not next to the block/dispenser, move towards it (using the go to movement heuristics).
  - Move around on the map (based on the safe exploration heuristic).

- Perform *skip* action.

2.4 Agent Knowledge

In this section, we cover the design of the knowledge stored in the agents' mental states. Generally speaking, the agents store and maintain knowledge about the map that is assumed be invariable (or alternatively: always perceivable). We consider invariable knowledge to be: the positions of goal cells, attached blocks, the agent's current position (relative to its starting position), positions visited by the agent, and the position of encountered agents from the team. Some of these involve communication between agents. The positions of blocks, dispensers and obstacles are only stored in the agent as long as they are within vision. The communication of our agents is described in Section 2.5.

The agent does not perceive a global view of the map via the environment, nor its own position on the map. Furthermore, random events and actions of other

agents can change the structure of the map over time. Due to this complexity, we do not attempt to build up an internal representation of the map. Unfortunately, this comes at the cost of efficient and meaningful movement on the map.

2.4.1 The Current Position

By keeping track of performed *move* actions, and checking for a potential failed action, the agent maintains information about its own current position. With the starting position of the agent as the center of origin, we maintain two values that represent the agent's position in a two-dimensional space. Moving in a direction, either north, east, south or west; results in incrementing or decrementing one of these values.

2.4.2 Visited Positions

The bookkeeping of visited positions is essential to avoiding that the agent repeatedly gets stuck, or does not make progress. When an agent performs a *move* action, in the next step the following information about the visited position is stored: the relative position of the agent, the current step in the simulation, and a flag for if the position is a goal cell. The position is stored relative to the initial position of each agent. That is, each agent is initially at $(x, y) = (0, 0)$. The relative position of each agent is updated based on successful *move* actions.

As the simulation progresses, the database of visited positions gains additional entries. Due to the way we utilize this, we are only interested in visited positions with respect to a specific subtasks. For instance, if the agent is trying to find a goal cell, it is not relevant which positions the agent visited in an attempt to find blocks. Therefore, we define a number of events that trigger a clearing of the agent's knowledge about visited positions:

- The agent has completed a subtask: Attached, detached or requested a block.
- We submitted a task. In our system, once a task is submitted, most agents will work on different matters.

The visited nodes are useful for steering the agent away from repeating the same movement patterns when they do not make progress. The idea is that visited positions are only relevant locally – at later points in time it may be relevant

to visit those positions again. Essentially, this means that the visited positions are only remembered for the duration of smaller subtasks. Intuitively, it seems non-optimal to remove knowledge that could help steer the agent away from dead ends that it has found earlier. However, the ever-changing structure of the map quickly invalidates this knowledge anyway.

2.4.3 Positions of Goal Cells

The positions of goal cells are assumed to be invariable throughout the simulation. Due to this assumption, once the agents store knowledge about the positions of goal cells, they are never removed.

The position of goal cells are stored relative to the position of the agents and are thus updated following successful *move* actions.

The agents learn about positions of goal cells either through perceiving them within their own vision or via communication with other agents.

2.4.4 Blocks, Dispensers and Obstacles

The information about blocks, dispensers and obstacles are only perceived when the agent is within vision. A *clear* event may remove blocks from the map, or they may be moved by other agents. Due to this, the positions of blocks is not maintained when outside of the agent's vision.

Obstacle positions could potentially be maintained by perceiving *clear* events and remove information about affected obstacles. The position of obstacles currently plays no part in any sort of route finding algorithm and we do not maintain this knowledge when outside of the agent's vision.

Dispensers are different from blocks and obstacles as their positions do not change during the simulation. In the current implementation, agents always go towards an available dispenser, if they do not have a block on each side, and if they are not trying to solve a task. Our agents will always prefer to go to the nearest known position of a block or dispenser. Thus to avoid the agent always going back to the same dispenser, we currently do not keep information of dispenser positions outside of the agent's vision.

Neither the position of blocks, dispenser or obstacles are shared between agents via communication. Since we do not keep and maintain their positions, it does

not make sense to share the information between agents – it should only be part of an agent's mental state when within vision.

2.4.5 Attached Blocks

Each agent keeps track of its own attached blocks with coordinates relative to its own position. The environment makes available any attached blocks in vision, but it is not immediately visible which agents the blocks are attached to. To make sure that the agent only keeps stored knowledge about the blocks attached to itself, we check for successful *attach* actions to insert the knowledge of a block being attached. Successful rotations update the stored coordinates accordingly. We always make sure that any knowledge about attached blocks is also perceivable in the environment – if not, the knowledge is removed. This is due to the fact that submitting tasks and *clear* events may invalidate the knowledge.

We will also briefly mention that current attached blocks are communicated between agents. This is used for devising plans to submit tasks. The details are covered in Section 2.5.3 and Section 2.7.2.

2.5 Agent Communication and Shared Knowledge

Sharing knowledge between agents by means of communication is essential for efficiently exploiting the multiple agents available. The environment presents a number of challenges in enabling effective agent communication. Also, the volatility of the scenario map does not suggest an easy way of building up a shared representation. Our current implementation could utilize more shared knowledge and communication, and it is something we hope to improve in the future.

Specifically for agent programming using GOAL, communication between agents are part of the core loop. One important aspect is that any messages sent in one step will only be available for processing by the receiving agent in the following step. This requires some deliberate implementation to make sure that the information received is up to date – in our implementation this is extremely relevant as we often share information that is relative to the current position of agents.

2.5.1 Encountering other Agents

The environment only gives information to agents about the position of other agents when within their vision. The agent is able to perceive which team an encountered agent belongs to, but no further identification is provided (i.e. the name of the encountered agent). To be able to identify which pair of agents that have encountered each other we apply the following: when two of our agents meet, they exchange information about what other objects they are able to identify within their vision. Only if they agree on everything in their shared vision, they acknowledge that they did in fact encounter each other. A check is performed to prevent two agents from mistakenly concluding that they encountered each other. We do this by checking that the given pair of agents agree on objects in their shared vision. Our initial implementation was solely based on the agents' relative position to each other, with no additional conditions, which yielded occasional false positives.

2.5.2 Goal Cells and Agent Positions

When two agents agree that they encountered each other, they exchange information about the positions of goal cells and other agents from the team. Each agent adds the shared information to its belief base relative to its belief about its own current position. Currently, the agents do not continue to share new information after encountering other agents.

When an agent successfully moves in a given direction, it informs other agents about which directed it moved in. This information is used by each agent to maintain the knowledge about positions of other agents.

2.5.3 Attached Blocks

We assign one of our agents as the *planning agent*. At each step of the simulation, each agent, that is not currently assigned to solve a task, sends a message to the planning agent containing a list of its currently attached blocks. The planning agent uses the received messages to (possibly) assign a task to a subset of the agents that sent messages. The details of the task planning assignment are covered in Section 2.7.2.

2.6 Agent Movement

The *Agents Assemble* scenario provides only partial vision of the map to agents, limited to a small area around each agent. Combining the knowledge of agents over time will provide more and more knowledge of the map. However, random *clear* events happen over time around the map that remove and randomly add new obstacles on part of the map. Other agents also have the ability to remove obstacles. As such, a usual route finding algorithm requires substantial alterations to be usable for the scenario. Due to the volatility of the map, such a route finding algorithm will necessarily have to support re-planning when the planned route is invalidated.

The above mentioned challenges for a route finding algorithm means that we have opted for a more naive implementation of agent movement. The overall strategy is to evaluate each of the (up to four) possible directions: north, east, south and west; and then select the direction which has the optimal heuristic value. When multiple directions share the same optimal value, a direction is selected pseudo-randomly (the current simulation step is used as seed). Our agent movement algorithm has five different variations:

- **Exploration** favors directions towards positions the agent has not visited recently.
- **Safe exploration** is similar to the above, but further favors directions that increase the distance to goal areas and other agents.
- **Go to** favors directions towards a given relative position and penalizes movement to recently visited positions on the map.
- **Task pattern** favors directions that realize a given task pattern and penalizes movement to recently visited positions on the map.
- **Detach** favors directions away from obstacles.

The choice of movement algorithm depends on the current strategy of the agent.

2.6.1 Evaluation Functions

As described above, each variation of the movement algorithm is distinguished by its heuristic function h. We will use h^+ to denote that a higher value is better and h^- when lower is better. Neither of the mathematical formulations are perfect in any sense, but give some approximation of the optimal choice.

2.6.2 Exploration

$$h^{+}_{exp}(d) = \sum_{visited} \begin{cases} \frac{|\Delta x(d)|+|\Delta y(d)|}{\Delta S^2} & \text{if } |\Delta x(d)| + |\Delta y(d)| \leq 30 \text{ and } \Delta S > 0 \\ 0 & \text{else} \end{cases}$$

where $\Delta x(d)$ and $\Delta y(d)$ are the differences in x and y coordinates between the visited position and the agent's position after performing *move* in direction d. ΔS is the number of steps since the position was visited.

2.6.3 Safe Exploration

$$h^{+}_{s-exp}(d) = h^{+}_{exp}(d) + \sum_{(x_t,y_t)\in P} c(x_t, y_t) * (|x_t| + |y_t|)$$

where P is a set of coordinates of goal cells and nearby agents in the team. x_t and y_t are coordinates relative to the agent's current position. $c(x_t, y_t)$ is a constant factor used for heavily favoring moving away from nearby agents.

2.6.4 Go To

$$h^{-}_{go-to}(d) = |\Delta x(d)| + |\Delta y(d)| + \text{size}(V_{p_d})$$

where $\Delta x(d)$ and $\Delta y(d)$ are the differences in x and y coordinates between the goal position and p_d is the agent's position after performing *move* in direction d. $\text{size}(V_{p_d})$ is the number of times position p_d has been visited recently.

2.6.5 Task Pattern

$$h^{-}_{pat}(d) = \text{size}(V_{p_d}) + \sum_{(x,y,t)\in(pat/att)} min\left\{|\Delta x| + |\Delta y|, |\Psi(d, x, y, t, \Delta x, \Delta y)|\right\}$$

where $\text{size}(V_{p_d})$ is the number of times p_d has been visited recently (p_d is the agent's position after performing *move* in direction d). The set pat/att contains the relative position and type of every block in the pattern excluding the blocks the agent itself is providing (has attached). The predicate $\Psi(d, x, y, t, \Delta x, \Delta y)$ gives the difference in x and y coordinates to every observed non-attached block of type t assuming a move in direction d.

2.6.6 Detach

$$h^+_{det}(d) = h^+_{exp}(d) + \sum_{(x,y)\in \text{obstacles}} |\Delta x(d)| + |\Delta y(d)|$$

where obstacles is the set of the positions of observable obstacles. $|\Delta x(d)|$ and $|\Delta y(d)|$ are the relative differences in x and y coordinates between the agent and the obstacle following a move in direction d.

2.7 Solving Tasks

This section describes how the agents solve tasks. We consider solving a task to consist of four parts: Collecting blocks, planning tasks to complete based on the collected blocks, executing task plans (assembling the pattern) and finally submitting the pattern.

2.7.1 Collecting Blocks

One core aspect of our strategy is to collect blocks before committing to any of the available tasks, and to only commit to tasks for which we already have the blocks to solve.

If an agent, that is not assigned to a task, and does not hold a block on each side, encounters a block or dispenser, it will generally try to go towards it. It will only ignore the possibility to collect the block(s) in case another from the same team is adjacent to it. This is to avoid race conditions for the same resource and improve efficiency. Dealing with this issue via communication would likely be a better approach, however.

In case the agent sees a dispenser or block, that is not occupied by another agent from the same team, the agent will rotate if necessary to ensure that a free attachment spot is available in the direction of the block or dispenser. Attaching blocks takes priority over requesting blocks from dispensers. The agent will repeatedly attach blocks on each of the four spots until they are all used.

In some cases the position of the block or dispenser may not allow the agent to attach from that angle due to its current attachments. This is not currently checked and avoided. In such a case, the agent is likely to enter a state of not

making progress – unless it is assigned to a task, or if it moves outside of vision of the block due to being penalized for going to similar positions repeatedly.

2.7.2 Task Planning

If the combined attachments of all agents are sufficient to solve a task the planning agent will compute a task plan for it. The task with the lowest reward (and thus, likely the easiest to complete) is selected. We select the easiest task based on the observed performance of the agents, and we only ever try to complete one task at a time. The logic for task planning supports computing multiple non-overlapping task plans, but we observed that the agents would be likely to obstruct each other. We only ever commit to a task that has some amount of steps available to complete the task before the deadline. For the contest, we set this to a minimum of 50 steps to complete the task. Later experimentation has yielded better results with a higher number. Due to the abundance of available tasks, the improved results seem logical. So far we have not conducted tests of the distribution of task completion times for our agents.

The task plan specifies how each agent provides part of the pattern including how it should be rotated. The expected completed pattern is computed for each agent relative to its own position in the aligned pattern. This is used to ensure that the agents align their attachments correctly. The task plan also specifies how agents should connect to each other once the pattern alignment step is complete. For each task, one of the selected agents is assigned to submit the task.

The task plans are rigid in the sense that two agents that provide the same block type cannot swap their respective sub-patterns. While there potentially could be some benefit in supporting this behavior, we do not consider it worth the effort to implement when considering other potential improvements to the agents.

2.7.3 Executing Task Plans

When a task plan is sent to the agents, the agents not involved will keep on moving around the map, but their heuristics for movement will penalize moving close to other agents. This is done in an attempt to avoid obstructing agents that are working on completing a task. Each of the involved agents will immediately detach any of the blocks that are no longer needed and rotate the remaining attachments to align with their part of the pattern for the task. The agent

responsible for submitting the task will move towards the nearest goal cell if its position is known (or scout for a goal cell if not). Once positioned at a goal cell, the agent will wait for the other agents to show up. If the other agents assigned to the task know the position of the agent responsible to submitting the task, they will move towards that agent. If not, they will methodically visit each of the known goal areas. If this fails, they will scout around the map. In most practical cases, the agents eventually learn of the position of the agent responsible for submitting the task.

We are not currently able to resolve situations where assigned agents are stuck. However, the task plan is discarded if an assigned agent is disabled due to a *clear* event. The idea is that if an agent is stuck the task plan is to be discarded, but our implementation does not work properly. As previously mentioned, we also do not utilize the *clear* action in any way currently, which could solve potential pathing problems. At any time, if the task deadline is exceeded, the task plan is deleted.

In case an agent assigned to the task finds its way to the agent responsible for submitting the task, waiting at a goal area, the agent will start to align itself to complete the pattern. This is achieved by the task pattern heuristic that favors directions that minimize the expected deviance from the pattern to submit. It should be noted that the agent responsible for submitting the task will try to position itself such that the other agents have room to align themselves to complete the pattern.

Once the pattern is complete, the agents connect their attachments and the task is submitted.

2.8 Evaluation of Matches

With a total of four participants, we played three matches against each of the three opponents. In the following, we evaluate the performance of our agents in each matchup. See Figures A.1 and A.45 in Appendix A for key statistics over the 500 steps of each match.

GOAL-DTU vs. TRG

In two of the simulations, we manage to complete a single task early on (Figures A.10 to A.12). Team TRG completes a single task in the first simulation,

but are unable to do so in the other simulations. TRG has a strategy where some of their agents defend goal areas by attempting to perform *clear* actions on our agents trying to complete tasks. Since part of our task execution plan is to assemble the pattern in the goal area, the strategy of team TRG denies a fair number of submits from our agents.

We experience problems with many agent getting stuck in every simulation. Around halfway through, we can usually observe that half of our agents are now stuck. This is especially detrimental if one of those agents is assigned to complete a task.

It seems that our greedy approach for collecting blocks, which causes the map to become even more convoluted, also causes serious issues for the agents of team TRG.

In the third simulation, the number of blocks we manage to collect stagnates (Figure A.6). This could be correlated with the number of *clear* events that is significantly higher (Figure A.15). While this presumably has no direct impact our score, it could be an indication towards our agents' ability to move around on the map.

See Figures A.1, A.2 and A.15 for all of the collected statistics in matches vs. TRG.

GOAL-DTU vs. FIT BUT

For this matchup we experienced issues with the agents. In the first simulation, our agents manage to assemble two patterns for tasks, but in one instance they seem to have a wrong pattern, and in the other instance they try to submit outside of a goal area (Figure A.22). At this point, we try to restart our agents, but they do not mange to make meaningful progress since our implementation is not robust in case of crashes (Figure A.19).

Also in the second simulation we have to attempt a restart, but to no avail. It seems our agents obstruct the map so severely that team FIT BUT has issues (Figure A.26). Before our agents crash, they are relatively close to assembling a pattern.

The story repeats itself in the third simulation, although team FIT BUT successfully complete multiple tasks (Figure A.27).

While we do not expect us to have been able to beat the agents of team FIT

BUT, we would likely have been able to complete a few tasks if the agents did not crash.

See Figures A.16 and A.30 for all of the collected statistics in matches vs. FIT BUT.

LFC vs. GOAL-DTU

We managed to complete tasks in each of the three simulations (Figures A.40 to A.42). Yet, as for other simulations, as the simulation progresses, the agents' ability to move around the map degrades (Figures A.34 to A.36). In comparison, it seems that the agents of team LFC make steady progress throughout the simulation (Figures A.40 to A.42). Another note about the agents of team LFC is there seems to be a correlation between the tasks they submit and the number of blocks they collect which suggest a different strategy (Figures A.34 to A.36).

We manage to complete more tasks in the second simulation (Figure A.41). By inspection of the map layout, there are two goal areas in the middle of map, not close to any obstacles. This is a lucky coincidence for our agents, as they often experience more problems when close to obstacles (maneuverability in confined spaces is more likely to degrade over time).

See Figures A.31, A.32 and A.45 for all of the collected statistics in matches vs. LFC.

2.9 Discussion

While our implementation achieved satisfying results, it can still be improved on several fronts. During the contest, we learned that the performance of the agents could be improved by tinkering with parameters, while other issues were due to technical difficulties.

2.9.1 Changes since the Contest

After analysis of the replays of our matches in the contest, we realized two possible improvements to the implementation. The first improvement is concerned

with the assumed time our agents need to complete a task. Through experimentation, we have learned that increasing the minimum amount of steps needed to complete a task improved the performance of the agents significantly. The value used for the matches in the contest often lead to agents missing task deadlines resulting in a lower score, when considering that a task with a later deadline could have been chosen instead.

Another issue occurs when agents detach blocks (for getting rid of blocks not needed for completing the assigned task). The agents will try to detach the unneeded blocks away from goal areas and obstacles. This is done to avoid that the agent potentially obstructs itself and other agents. By increasing the minimum distance there should be to goal cells and obstacles when detaching a block, we observed improved performance.

2.9.2 Technical Issues during the Contest

During the contest, we experienced a number of issues during the simulations that we had not encountered before. Unfortunately, for some of the matches this made our agents break down completely, practically leaving us with no way to continue. Since we did not experience this before, our agents are not very robust in the sense that they are not well-suited for restarts during the simulation in case of crashes.

One of the issues we experienced occurs when the steps are performed very rapidly, at which point it seems as if the GOAL execution and the server get out of sync. We did not experience this problem earlier since in our testing, there were always agents not performing actions, thus using the full server timeout for each step. Experimentation following the contest shows that the problem is not related to connection issues and will have to be investigated further. We find that two seconds for each step prevents the issue.

2.9.3 Known Problems and Bugs

In our implementation, we have discovered a number of problems over time, and there are still some unresolved bugs to fix.

One problem is related to agents getting stuck. Often in simulations, we will experience that one or more of our agents end up getting stuck. This could be due to a number of random *clear* events, or potentially the map has disconnected parts. One of the major issues with this is that we have not been able to

implement a way for agents to deduce that they are in fact stuck, or that some agents are unable to reach each other. Another problem is that we do not utilize the *clear* action to help the agents become unstuck.

Lastly, we have experienced problems when agents assigned to a task visit goal areas in search for the agent responsible for submitting the task. Due to an unresolved bug, the agents will not properly scout all the goal areas, but tend to always stay in the same area. This obviously means that we will never be able to submit unless the problematic agents learn about the position of the agent responsible for submitting the task.

2.9.4 Improvements

There are a number of directions to take in terms of improving the implementation. We will consider some of our high-level ideas to improve the performance of the system by targeting some of our weaknesses.

Our agents are universal in the sense that every agent is based on exactly the same logic rules. One way to improve the performance, could be to assign different roles to agents, or to assign agents into smaller teams that move together. Examples of roles could be agents that explore the map, some that request and collect blocks and some that complete tasks using the collected blocks.

Another weakness of our agents is poor movement. Since we do not build an internal representation of the map during the simulation, our agents often move blindly. We expect substantially improved performance if we are able to make the agents better at moving around the map, for example by building up such an internal map representation (which could then be shared among agents). The initial reason not to attempt building up a map representation, is the complexity of the map being dynamic.

Another problem arises from our greedy approach to collecting blocks. Since each agent always tries to collect one block on each side, movement around the map becomes much harder afterwards. Furthermore, we do not consider the possibility that an agent may be able to move past narrow corridors by rotating its attachments, or potentially even moving the blocks past corridors a few at a time.

The last obvious improvement is to implement some logic to perform *clear* actions. Multiple *clear* events are likely to make it difficult to move around the map. Getting rid of obstructions is exactly one of the purposes of the *clear* action. It can be used to reconnect parts of the map that has been disconnected

completely, and to save valuable time by creating shortcuts through obstacles.

Lastly, there are number of technical improvements that we would like to implement. Since it was impossible to monitor the agents live during the simulations of the contest, it would have been helpful to have better output (in the console) about the behavior and progress of the agents. Furthermore, we would like to increase the robustness of the agents in case of crashes such that they can be restarted and still make progress.

2.10 Conclusion

We have provided an overview of the multi-agent system that the GOAL-DTU team developed for the Multi-Agent Programming Contest 2019. We have explained our choice of the GOAL programming language; we have also described the main strategy of our agents and how they execute that strategy. This year was the first iteration using the *Agents Assemble* scenario, and we have developed our implementation from scratch using GOAL. Our implementation features a universal agent type in which each agent is based on the same set of logical rules.

The strengths of our system are the flexible nature of our agents. Our agents always react to the current state of affairs and do not rely heavily on predefined plans to reach their goal of completing tasks. The weaknesses are primarily the agents' poor movement around the map and rigidity in the way tasks are assigned and submitted where stuck agents have a severe negative impact on the performance of the system.

Finally, we have described how to improve the system by coming up with ideas that target its weaknesses. Some of these potential improvements are related to minor issues and bug fixes while other potential improvements require designing and refactoring large parts of the system.

In conclusion, we are satisfied with the performance of our system, ending at a 3rd place in the final rankings, when considering that we have built the system from scratch. We consider our current implementation a good platform to built on for future iterations of the *Agents Assemble* scenario.

Acknowledgements

We thank Tobias Ahlbrecht, Asta Halkjær From, Benjamin Simon Stenbjerg Jepsen, John Bruntse Larsen and Simon Rumle Tarnow for discussions.

This chapter is a published article. See Section 1.6 for details.

Chapter 3

Towards Verifying a Blocks World for Teams GOAL Agent

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.

3.1 Introduction

It is of key importance to provide assurance of the reliability of multi-agent systems [28]. However, demonstrating the reliability of such systems remains a challenge due to their often complex behavior, usually exceeding the complexity of procedural programs [85].

Popular approaches to programming agents have been inspired by the agent-oriented programming paradigm [72]. Examples include JADE for the Java-platform and GOAL that is inspired by logic programming [6, 36]. A notable

difference between the two is that GOAL implements a BDI model [67] whereas JADE is a middleware that facilitates development of multi-agent systems under the FIPA standard [65]. 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 [22] 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 [51]. 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 3.2 relates to existing work in the literature. Section 3.3 introduces GOAL and the BW4T environment. Section 3.4 describes a proof theory for GOAL. Section 3.5 presents a transformation method from GOAL programs to agent logic. Section 3.6 describes how to verify agents in the agent logic. Finally, Section 3.7 gives concluding remarks.

3.2 Related Work

The work in this paper relates to our work on formalizing GOAL and the verification framework in a proof assistant [44]. 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 [47], 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 [56], 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 mechanized in the higher-order logic theorem prover Isabelle/HOL [58, 61].

In [2] theorem-proving techniques are applied to verify correctness properties of simpleAPL agents (simpleAPL is a simplified version of 3APL [35]). 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. [24] has worked towards techniques for analyzing implemented multi-agent system platforms based on BDI. With a starting point in model checking techniques for AgentSpeak [12], these are extended to other languages including GOAL. From a recent review of logic-based approaches for multi-agent systems by [17], it is clear that model checking has retained its position as the primary tool for practical verification of multi-agent systems.

3.3 Background

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

3.3.1 GOAL Programming Language

Below we give a brief introduction to the GOAL programming language; for further details we direct the reader to [22, 39].

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 [68].

3.3.2 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 3.1 shows the relevant subset of the available percepts and actions of the BW4T environment.

Following the Russell & Norvig categorization of environments [70], BW4T can for our configuration be categorized as a single-agent, partially observable, deterministic environment. This categorization plays an essential part in enabling modeling 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 3.1 shows the initial state of the example map where the goal is to collect a single red block (indicated in the bottom left corner).

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.

Table 3.1: Relevant percepts and actions for our agent in the BW4T environment.

3.3.3 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 [25], 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.

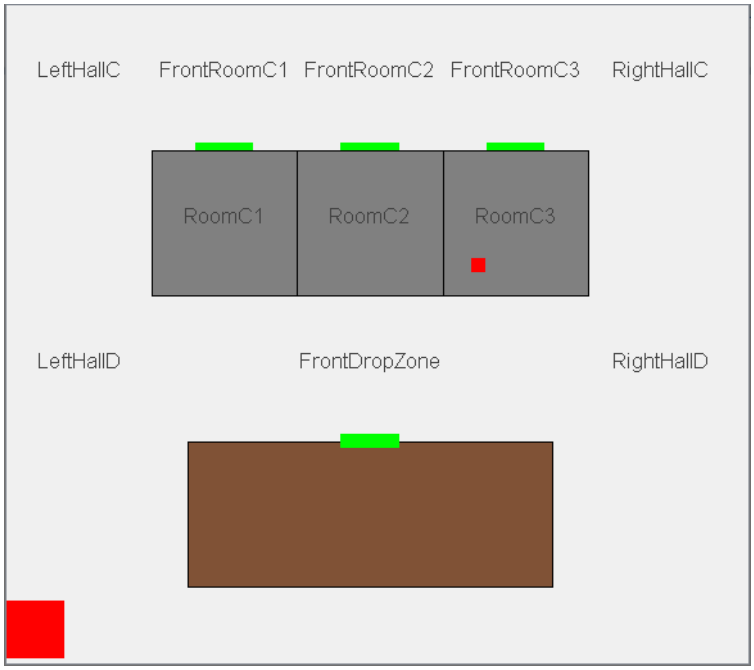

Figure 3.1: The example map in the initial state.

3.4 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.

3.4.1 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

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:	$\dfrac{\{\varphi_1\}\ a\ \{\psi\} \qquad \{\varphi_2\}\ a\ \{\psi\}}{\{\varphi_1 \vee \varphi_2\}\ a\ \{\psi\}}$

Table 3.2: The proof rules of our Hoare system.

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 3.5. 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.

3.4.2 Proof System

We now consider a Hoare system for GOAL. The proof rules are as defined in [22] and are listed in Table 3.2.

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 [22] in which the Hoare system is proved sound and complete with respect to the semantics of Hoare triples on mental state formulas.

3.5 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 modeled 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.

3.5.1 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 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 postconditions 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:

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

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 $enabled(goTo(\mathrm{X}))$. 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:

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

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:

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

The shown instantiation of the effect axiom will be useful for the correctness proof we go through in Section 3.6. While not apparent from the example, we have a special case for the *goTo* action when going back to 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.

3.5.2 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 modeling 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.

3.6 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**:

$$\frac{\varphi \,\mathbf{ensures}\, \psi}{\varphi \mapsto \psi} \qquad \frac{\varphi \mapsto \chi \;\; \chi \mapsto \psi}{\varphi \mapsto \psi} \qquad \frac{\varphi_1 \mapsto \psi \;\; \ldots \;\; \varphi_n \mapsto \psi}{(\varphi_1 \vee \ldots \vee \varphi_n) \mapsto \psi}$$

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 $\{\varphi \wedge \neg\psi\}\ a'\ \{\psi\}$.

3.6.1 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:

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

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:

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

The full proof outline is a sequence of **ensures** statements that lead to the desired mental state as shown in Figure 3.2. This sequence is explicitly stated and of course ties into the belief that there exist actions that will satisfy each step.

(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 3.2: Proof outline consisting of five steps.

In the proof outline in Figure 3.2 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 to prove, for every action a:

$$\begin{aligned}
&\{\mathsf{B}color(b_a, red, r_1) \land \mathsf{B}in(r_0) \land \neg\mathsf{B}holding(b_a) \land \mathsf{G}collect(red) \\
&\quad \land \neg(\mathsf{B}color(b_a, red, r_1) \land \mathsf{B}in(r_1) \land \neg\mathsf{B}holding(b_a) \land \mathsf{G}collect(red))\} \\
&\ a \\
&\{\mathsf{B}color(b_a, red, r_1) \land \mathsf{B}in(r_0) \land \neg\mathsf{B}holding(b_a) \land \mathsf{G}collect(red) \lor \\
&\ \mathsf{B}color(b_a, red, r_1) \land \mathsf{B}in(r_1) \land \neg\mathsf{B}holding(b_a) \land \mathsf{G}collect(red)\}
\end{aligned}$$

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

$$\begin{aligned}
&\{\mathsf{B}color(b_a, red, r_1) \land \mathsf{B}in(r_0) \land \neg\mathsf{B}holding(b_a) \land \mathsf{G}collect(red) \\
&\quad \land \neg(\mathsf{B}color(b_a, red, r_1) \land \mathsf{B}in(r_1)\ land\neg\mathsf{B}holding(b_a) \land \mathsf{G}collect(red))\} \\
&\ a' \\
&\{\mathsf{B}color(b_a, red, r_1) \land \mathsf{B}in(r_1) \land \neg\mathsf{B}holding(b_a) \land \mathsf{G}collect(red)\}
\end{aligned}$$

Each of these Hoare triples are proved by applying the proof rules of Table 3.2. 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 in Appendix B.

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 modeling systems with more than just a single agent. There have been efforts towards agent logics for multiple agents, such as by [16], but 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 [55]. 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 Jørgen Villadsen for comments on drafts of this paper.

This chapter is a published article. See Section 1.6 for details.

Chapter 4

On Using Theorem Proving for Cognitive Agent-Oriented Programming

Demonstrating reliability of cognitive multi-agent systems is of key importance. There has been an extensive amount of work on logics for verifying cognitive agents but it has remained mostly theoretical. Cognitive agent-oriented programming languages provide the tools for compact representation of complex decision making mechanisms, which offers an opportunity for applying a theorem proving approach. We base our work on the belief that theorem proving can add to the currently available approaches for providing assurance for cognitive multi-agent systems. However, a practical approach using theorem proving is missing. We explore the use of proof assistants to make verifying cognitive multi-agent systems more practical.

4.1 Introduction

Reliability is of key importance for software systems in general and multi-agent systems (MAS) and cognitive MAS (CMAS) in particular [28]. Cognitive multi-agent systems are systems that consist of agents that use cognitive notions such as beliefs and goals to decide on their next action. Dedicated cognitive agent programming languages have been developed for engineering these systems. Be-

cause these languages have incorporated these high-level concepts of beliefs and goals as first-class citizens they can rather compactly represent quite intricate decision making mechanisms. However, demonstrating the reliability of such CMAS remains very challenging because of the complex behavior these mechanisms are able to generate. We often observe complex behavior patterns of agents that exceed those of traditional procedural programs [85].

Testing methods for CMAS have been explored by [21, 54], but testing alone is unlikely to provide the levels of assurance required by stakeholders. To provide this level of assurance for complex CMAS, formal verification techniques are needed for demonstrating their reliability. The fact that several agent languages have been specified by means of a formal semantics, e.g. 3APL, GOAL [35, 36], and AgentSpeak(L) languages such as Jason and JaCaMo [8, 10, 66], provides a suitable starting point for developing these techniques. The initial developments of BDI logics such as by [67] and [19] did not connect well with the more practically oriented work done in engineering and programming of agents such as [35]. These logics were significantly more complicated which provided a barrier, and only little progress was made to close the gap it created. Although work such as [9, 52] has provided more practical tools to demonstrate reliability using a model checking approach, we notice that the work on verification logics has remained mostly theoretical thus far.

The success of state-of-the-art proof assistants has been demonstrated in the literature for more traditional software development paradigms different from CMAS. They have proved quite successful for verification of various software (and hardware) systems [69]. One major branch is based on simple type theory (higher-order logic) with prime examples being HOL Light, HOL4 and Isabelle/HOL [32, 58, 73]. Another major branch contains those based on dependent type theory such as Agda, Coq and Lean [3, 7, 75]. Built from a small, trusted kernel proof assistants provide tools for ensuring reliability of systems that are also trustworthy [63]. Arguably, these systems remain accessible mostly for specialists and have been applied in practice to relatively limited cases. The answer to why this is the case seems obvious: because only then it pays off. Applying it to much larger programs is too complicated and requires too much effort. But we believe that here there is actually a benefit of CMAS that still needs to be explored: CMAS are typically much more compact programs, and agent programming languages are typically very high-level programming languages that introduce high-level concepts such as beliefs and goals. At least at first sight this appears to make them more suitable candidates for a theorem proving approach.

Given the success of proof assistants in other areas, there should be something for us here to explore. To the best of our knowledge, there is no work thus far that has looked into the viability of a theorem proving approach. More

specifically, we aim to explore the use of proof assistants as a practical alternative for demonstrating the reliability of CMAS.

In this paper, we want to explore a practical approach as an answer to our main research question: *how can a theorem proving approach contribute to reliable engineering of cognitive agent systems?*

We argue that proof assistants can be useful and play an important role in verifying cognitive multi-agent programs. To support our argument, we use the GOAL agent programming language as our primary example. We do so partly because the core of this language captures the core concepts of cognitive agent programming but also is relatively simple (when we leave out more advanced concepts for structuring agent programs such as modules) and a formal semantics is available which facilitates the process of formalization in a proof assistant.

The paper is structured as follows. Section 4.2 highlights some of the relevant existing work in the literature. Section 4.3 gives a brief introduction to the basic concepts of the programming language GOAL. Section 4.4 discusses the challenge of ensuring reliability of CMAS and compares the current available approaches for demonstrating reliability. Section 4.5 explores how a theorem proving approach can contribute to demonstrating reliability. Section 4.6 serves as an appetizer for early work on formalizing GOAL in Isabelle/HOL. Finally, Section 4.7 makes some concluding remarks.

4.2 Related Work

Agent systems, especially in the multi-agent setting, are notoriously hard to test. The findings of [85] show that belief-desire-intention (BDI) agents indeed have a large number of paths through the program — something that has often been presumed. In the case of white box testing, covering all paths thus becomes increasingly impossible. Perhaps surprisingly, it is also found that adding failure handling to agents significantly increases the size of the behavior space.

Traditionally, temporal logic has been the preferred language for specification of agents and correctness properties, but it falls short when considering e.g. error-prone environments and capturing that agents succeed where possible. In such cases the straightforward statement that the agent will eventually succeed is too strong and not provable. [84] suggest an approach that combines testing and formal verification as neither in practice succeeds to obtain assurance of the reliability of the system.

The focus of formal verification of CMAS has mostly been on testing with somewhat limited logical languages: [52] suggest an approach using a model checker on top of the program interpreter. The model checker is implemented for the interpreter of the GOAL language, but the proposed framework is generally applicable to other agent programming languages as well. [54] propose an automated testing framework for automatically detecting failures in cognitive agent programs. Although the work shows promise, more work is needed particularly towards localization of failures.

[53] explore an approach applying the concept of omniscient debugging to cognitive agent programs. The work is based on the idea of reversing the program's execution rather than trying to debug the program by reproducing failure in a rerun. In agent programs this becomes especially useful as they usually have both non-deterministic aspects and rely on agent environments both of which make the reproduction of the circumstances of failure difficult.

Some work on theorem proving was done by [2] and in particular by [71] which explored verification of agents specified in Cognitive Agents Specification Language (CASL). However this work was focused mostly on the specification level and thus did not connect well with agent-oriented programming (AOP). As such, it still left a gap in terms of practical applicability for CMAS.

The present paper closely relates to our work in [44] where the formalization of GOAL in Isabelle/HOL (to be presented in Section 4.6) is discussed in details; [43] explores a method for transforming GOAL agents' program code to an agent logic of a verification framework — unlike this paper, a theorem proving approach is not considered.

4.3 The Programming Language GOAL

We use GOAL as our primary example of a cognitive AOP (c-AOP) language. This section briefly introduces the basic concepts of GOAL. For a more detailed overview of the GOAL agent programming language we refer to [36, 37, 39].

The language GOAL is an agent-oriented programming language based on the BDI logic that draws inspiration from concurrent programming in general and from the programming language UNITY in particular [56]. However, unlike UNITY, the basic concepts of the GOAL language are cognitive concepts such as beliefs and goals instead of simple assignments. GOAL uses a declarative notion of goals that agents use to decide their next action.

An agent's beliefs and goals are drawn from a classical, propositional language and are stored in a belief base and goal base, respectively. A belief base Σ and goal base Γ constitute an agent's mental state $\langle \Sigma, \Gamma \rangle$. The constraints imposed on mental states are a consequence of the blind commitment strategy that agents use by default [68]:

- Σ is consistent, i.e. does contain a contradiction,
- for all $\gamma \in \Gamma$:
  - γ is not entailed by the agent's beliefs, and
  - γ is satisfiable.

Formulas over mental states are used to express conditions on beliefs and goals. The language of mental state formulas is formed by Boolean combinations of the belief and goal modalities B and G, respectively. It may be helpful to think of these modalities as queries for inspection of the mental state:

- $B\Phi$: is Φ entailed by the belief base Σ?
- $G\Phi$: is Φ a goal (or subgoal) of the goal base Γ?

Besides these more complex notions for defining and reasoning about an agent's state, the GOAL language provides rules and capabilities (or basic actions) for deriving an agent's choice of action from its mental state. In contrast to other agent programming languages, as the goals of an agent in GOAL are declarative, they specify (conditions on) states the agent wants to achieve and not how to achieve them. The specification of rules provides means to program the agent such that it rationally selects actions for achieving its goals. As such, the language is built on the core principle that agents always derive their choice of action from their current beliefs and goals.

4.4 Ensuring Reliability

The paradigm of c-AOP draws many parallels with concepts of concurrent programs which are notoriously difficult to both test and formally verify. The primary reason is their complex behavior patterns and by extension the number of possible paths through the program. Consequently, covering every path of the program through testing is practically impossible and any proof attempt will explode in size.

Programming concept	Debugging	Automated testing	Model checking	Theorem proving
Knowledge representation	✓	✓	✓	⋆
Cognitive concepts	✓	✓	✓	⋆
Decision rules	✓	✓	✓	⋆
Multiple agents	✓	✓	✓	⋆
Environments	✓	×	×	⋆

Table 4.1: Programming concepts of multi-agent systems and possible approaches for demonstrating their reliability.

For languages that have a formal semantics, such as GOAL, formal techniques can be used, but the focus has been mostly on limited logical languages and model checking. Although these works have shown that they can be applied in practice, the work on verification logics has remained mostly theoretical thus far, and there has not been much work to show that proof assistants can contribute to demonstrating reliability of CMAS.

We want to pursue the idea that a proof assistant can contribute to reliability of CMAS. [17] recently reviewed logic-based technologies for MAS and provide a nice overview of the available, and actively developed, agent technologies including means of agent verification. Notably, while model checking is highly represented, the literature review reveals no mention of theorem proving nor proof assistants. In conclusion, it seems that these tools and methodologies have not been adopted in the MAS community. The question remains why this is the case. At the time MAS became trending, model checking had manifested itself as an effective formal technique for verification of software [18]. As such, adapting model checking to MAS seemed an obvious choice of direction for verification. Going back to the initial work on BDI agent logics, a lot of the work was promising. However, the practical applications were not explored further, possibly due to the need for automatic tools to reduce the effort of proof, and working with such tools may have seemed a daunting task. However, the AOP approach does not require as complex a logic as the original work in BDI and intention logics [38].

Some of the typical challenges for engineering CMAS reliably are: knowledge representation (KR), cognitive concepts such as beliefs and goals, decision rules, multiple agents, environments (the external world). Each of these contributes to the complexity of demonstrating the reliability, either by adding to the complexity or making it easier. We have listed the key programming concepts of MAS in Table 4.1 and an indication of which approaches we believe currently are state-of-the-art (✓) or not well established (×). Theorem proving for MAS

is something we are currently exploring ($\star$).

Debugging approaches remain the most versatile as they cover all CMAS concepts, but their weakness is that debugging only truly makes sense once an error has been observed — they do not inherently aid us in finding (obscure) error states. To overcome this, it often makes sense to devise a number of automated tests, e.g. by checking that certain properties hold before and after execution of modules. Outside of the immediate difficulties of devising testing schemes, a great difficulty lies in deterministic reproduction of error states when connected to external environments. The highest degree of reliability is currently achieved by applying model checking techniques. However, a potential pitfall is the often necessary introduction of assumptions to reduce the size of the state space.

4.5 A Theorem Proving Approach

In this section, we want to advocate an approach using theorem proving to ensure reliability of MAS. We already mentioned how the use of proof assistants has shown success in other applications of software.

We are particularly interested in what could be gained from a theorem proving approach for c-AOP for which we see some particular benefits: the cognitive concepts define primarily single agents and only in second instance the multi-agent level because agents communicate about their beliefs and goals. Especially the proof assistants based on higher-order logic are appealing. Here, we focus on Isabelle/HOL. Primarily because this is the proof assistant we have most experience working with. The strengths of Isabelle/HOL include first and foremost a powerful automatic proof search, including an automatic search for counterexamples (Isabelle will inform us that a particular proof goal cannot be solved due to the found counterexample) [62]. Furthermore, Isabelle facilities a readable language for structured proofs: Isar. The practicality of Isabelle rests on its LCF-style foundation: every proof goes through the kernel using abstract types to ensure its soundness [64].

Much of the work with formal techniques for verification of CMAS has been with limited logical languages. A reason for the lack of experimentation with higher-order logic (HOL) languages could be that finding the proper level of abstraction is difficult: the verification of agents is clearly bound to the context in which they are deployed. If we make our models too general, verification by proof will likely fail. At the other end of the spectrum, if we fail to sufficiently generalize the proof goal, the proof result may be too weak. It is clear that using a proof assistant will not inherently provide the answer to how a HOL-based

model of CMAS could look like. Yet still, the practicalities should make the approach more feasible from an engineering perspective. We are interested to see if a new and fresh perspective could emerge through the use of HOL, and what a general sketch of a HOL-based approach using theorem proving could look like.

We need to consider how to demonstrate the effectiveness of a theorem proving approach using proof assistants. The nature of verification can be very rigorous: the proof attempt either fails or succeeds. One way to alleviate this absoluteness is to compare the approach to similar approaches such as model checking: where do we achieve the highest degree of precision in the model, which approach produces the strongest theoretical results, how efficiently do we reach these results and how much effort is required to obtain them?

4.6 Formalizing GOAL in Isabelle/HOL

To demonstrate the feasibility of the suggested approach, we have started work on a formalization of the GOAL agent programming language and a verification framework. We take as our starting point and frame of reference the work by [22].

In the following we will illustrate our suggested theorem proving approach using Isabelle/HOL. For further details see [44].

The Isabelle files are publicly available online:

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

The theory file *Gvf_PL.thy* formalizes the propositional logic that serves as foundation. The theory *Gvf_GOAL.thy* imports *Gvf_PL.thy* and formalizes GOAL (up to the point of proving soundness of the presented proof system).

The Isabelle/HOL formalization is around 400 lines of code in total and loads in less than a second on a MacBook Pro with a 2.4 GHz 8-core i9 processor and 32 GB memory.

The website also links to demonstrations related to other work we have done on verification of GOAL agents [43].

In Isabelle/HOL expressions to be parsed within the context of the (embedded)

higher-order language are encapsulated in the so-called "cartouches" ‹ · · · ›. Note that they generally can be omitted for atomic expressions.

Cognitive concepts of agents are modeled through mental states. Mental states consists of a belief base and goal base that can be set up as a datatype over formulas of propositional logic (the type Φ_L):

type-synonym *mst* = (Φ_L *set* × Φ_L *set*)

The command **type-synonym** introduces a new type name as an abbreviation.

Certain constraints apply: not every tuple of sets of formulas qualify as mental states. We think of the declarative goals as achievement goals which is (partially) captured by the following definition:

definition *is-mst* :: *mst* ⇒ *bool* (∇) **where**
∇ *x* ≡ *let* (Σ, Γ) = *x* *in* Σ ¬⊨$_C$ ⊥$_L$ ∧ (∀γ∈Γ. Σ ¬⊨$_C$ γ ∧ {} ¬⊨$_C$ ¬$_L$ γ)

The command **definition** introduces a new definition. Unlike abbreviations, definitions are not automatically expanded by Isabelle which is helpful when a higher level of abstraction is desired. In our definition above, the type is ‹*mst* ⇒ *bool*›: given a mental state, an evaluation of the expression will result in a Boolean value. We introduce ∇ as a shorthand for the definition. The variable *x* is a pair of sets, as we defined earlier, and *let* (Σ, Γ) = *x* *in* ... introduces local variables Σ and Γ in ... by unfolding the *x* definition.

We have detached the type for the mental states (*mst*) from the constraints (*is-mst*). This gives some extra footwork: we need to introduce the definition into statements concerning mental states. On one hand we may need the definition as an assumption for a proof to go through; on the other hand we may need to prove that alterations preserve the mental state properties.

For reasoning about mental states, a language is embedded into Isabelle that allows for inspection of mental states through the belief and goal modalities, *B* and *G*, respectively:

datatype Φ_M =
B Φ_L | *G* Φ_L | *Neg* Φ_M (¬$_M$) | *Imp* Φ_M Φ_M (**infixr** ⟶$_M$ *60*) |
Dis Φ_M Φ_M (**infixl** ∨$_M$ *70*) | *Con* Φ_M Φ_M (**infixl** ∧$_M$ *80*)

The command **datatype** introduces a new datatype. Each option has a constructor followed by one or more input types. Multiple options are divided by

the special | character. The constructors may be recursive as is the case for the Boolean operators. The keywords **infixl** and **infixr** signify that the following shorthand is left- or right-associative, respectively. The numbers define the order of precedence.

Next we define the semantics of mental state formulas. The belief modality requires that the formula is entailed by current beliefs. The goal modality is more complicated: we either require that the formula is a goal or a subgoal (not entailed by current beliefs):

primrec *semantics* :: *mst* $\Rightarrow$ Φ_M $\Rightarrow$ *bool* (**infix** $\models_M$ *50*) **where**
$M \models_M (B\ \Phi) = (let\ (\Sigma, \text{-}) = M\ in\ \Sigma \models_C \Phi)$ |
$M \models_M (G\ \Phi) = (let\ (\Sigma, \Gamma) = M\ in$
$\Phi \in \Gamma \vee \Sigma \neg\models_C \Phi \wedge (\exists \gamma \in \Gamma.\ \{\} \models_C \gamma \longrightarrow_L \Phi))$ |
$M \models_M (\neg_M\ \Phi) = (\neg\ M \models_M \Phi)$ |
$M \models_M (\Phi_1 \longrightarrow_M \Phi_2) = (M \models_M \Phi_1 \longrightarrow M \models_M \Phi_2)$ |
$M \models_M (\Phi_1 \vee_M \Phi_2) = (M \models_M \Phi_1 \vee M \models_M \Phi_2)$ |
$M \models_M (\Phi_1 \wedge_M \Phi_2) = (M \models_M \Phi_1 \wedge M \models_M \Phi_2)$

The command **primrec** defines a primitive recursive function. We introduce infix syntax ($\models_M$) similar to textbook definitions. The semantics for Boolean operators can be defined by using Isabelle's built-in operators.

As a slight deviation from [22], we have baked the subgoal property into the semantics of the language. In the original work, the goal base is defined such that it includes all subgoals. This means that the goal base is an infinite set of formulas. We have opted for a constructive approach.

Now that we have defined the language and its semantics, we turn to prove some interesting properties of the goal modality. The following lemma justifies to call G a logical operator:

lemma *G-properties*:
shows
$\neg\ (\forall \Sigma\ \Gamma\ \Phi\ \psi.\ \nabla\ (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G\ (\Phi \longrightarrow_L \psi) \longrightarrow_M G\ \Phi \longrightarrow_M G\ \psi)$
and
$\neg\ (\forall \Sigma\ \Gamma\ \Phi\ \psi.\ \nabla\ (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G\ (\Phi \wedge_L (\Phi \longrightarrow_L \psi)) \longrightarrow_M G\ \psi)$
and
$\neg\ (\forall \Sigma\ \Gamma\ \Phi\ \psi.\ \nabla\ (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G\ \Phi \wedge_M G\ \psi \longrightarrow_M G\ (\Phi \wedge_L \psi))$
and
$\{\} \models_C \Phi \longleftrightarrow_L \psi \longrightarrow \nabla\ (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G\ \Phi \longleftrightarrow_M G\ \psi$

The command **lemma** instantiates a proof with the given proof goal(s) that are to be solved. In our case, we have stated four individual goals, each stating a

different property of the G modality. The proof details are omitted. We may later use the result of the lemma by referencing its name *G-properties*.

The properties of the modalities form the basis for an inductively defined proof system for mental state formulas:

inductive *derive* :: $\Phi_M \Rightarrow$ *bool* $(\vdash_M)$ **where**
R1: *conv* (*map-of* T) $\varphi' = \varphi \Longrightarrow \{\#\} \vdash_C \varphi' \Longrightarrow \vdash_M \varphi$ |
R2: $\{\#\} \vdash_C \Phi \Longrightarrow \vdash_M (B\ \Phi)$ |
A1: $\vdash_M (B\ (\Phi \longrightarrow_L \psi) \longrightarrow_M (B\ \Phi \longrightarrow_M B\ \psi))$ |
A2: $\vdash_M (\neg_M\ (B\ \bot_L))$ |
A3: $\vdash_M (\neg_M\ (G\ \bot_L))$ |
A4: $\vdash_M ((B\ \Phi) \longrightarrow_M (\neg_M\ (G\ \Phi)))$ |
A5: $\{\#\} \vdash_C (\Phi \longrightarrow_L \psi) \Longrightarrow \vdash_M (\neg_M\ (B\ \psi) \longrightarrow_M (G\ \Phi) \longrightarrow_M (G\ \psi))$

The command **inductive** introduces an inductive definition. The use of $\Longrightarrow$ signifies that side conditions apply (the result is to the far right).

In the inductive definition above, $\vdash_C$ denotes derivability in classical logic. The first condition of *R1* enforces that φ' and φ are structurally equivalent. As such, *R1* transfers any proof rules for classical logic via $\vdash_C$. *R2* captures that agents believe tautologies to be true. *A1-A5* are the axioms and have no side conditions.

We can prove the proof system sound with respect to the semantics of mental state formulas (again the proof is omitted, but we note that it in its current form requires around 80 lines):

theorem *derive-soundness*:
assumes $\nabla\ M$
shows $\vdash_M \Phi \Longrightarrow M \models_M \Phi$

The command **theorem** is internally similar to **lemma**. The distinction is reminiscent of writing formal proofs on paper, i.e. **theorem** signals an important result.

In summary, we acknowledge that our work is still in its early stages. Nevertheless, we think that the intermediate results show good promise that a full formalization of the GOAL programming language and the verification framework is possible.

4.7 Conclusion

We have considered reliability of cognitive multi-agent systems (CMAS) which consist of agents that use cognitive notions such as beliefs and goals.

The work on finding and improving approaches to demonstrating reliability of CMAS has in the multi-agent system community been dominated by model checking, debugging and testing. While all of these approaches have shown to be effective and practical, efforts towards new and better strategies are ongoing.

We have observed that while much of the early work on agent and verification logics showed promise, it has remained mostly theoretical and with limited practicality. We question if the reason could be due to the trends at the time work with CMAS gained traction, and not because the approach is inferior to others.

The early work on theorem proving as an approach to demonstrate reliability of CMAS has left a gap between theory and practice. We find that proof assistants provide the practical tools to close this gap, and that they can contribute to demonstrating reliability, primarily by means of powerful automation for proof search.

To illustrate the feasibility of the suggested approach, we have started work on formalizing the GOAL agent programming language in the Isabelle/HOL proof assistant. Our early findings show good promise for the approach.

Acknowledgements

We would like to thank Asta Halkjær From for Isabelle related discussions and for comments on drafts of this paper.

This chapter is a published article. See Section 1.6 for details.

Chapter 5

Machine-Checked Verification of Cognitive Agents

The ability to demonstrate reliability is an important aspect in deployment of software systems. This applies to cognitive multi-agent systems in particular due to their inherent complexity. We are still pursuing better approaches to demonstrate their reliability. The use of proof assistants and theorem proving has proven itself successful in verifying traditional software programs. This paper explores how to apply theorem proving to verify agent programs. We present our most recent work on formalizing a verification framework for cognitive agents using the proof assistant Isabelle/HOL.

5.1 Introduction

In the deployment of software systems we are often required to provide assurance that the systems operate reliably. Cognitive multi-agent systems (CMAS) consist of agents which incorporate cognitive concepts such as beliefs and goals. Their dedicated programming languages enable a compact representation of complex decision-making mechanisms. For CMAS in particular the complexity of these systems often exceeds that of procedural programs [85]. This calls for us to develop approaches that tackle the issue of demonstrating reliability for CMAS.

The current landscape reveals testing, debugging and formal verification such

as model checking as the primary approaches to demonstrating reliability for CMAS [17]. This landscape has primarily been dominated by work on model checking techniques and exploration of better testing methods [9, 52]. However, if we take a deeper look, we find promising work on theorem proving approaches [2, 71]. Combining testing and formal verification has also been suggested as their individual strengths and weaknesses complement each other well [84].

A proof assistant is an interactive software tool which assists the user in developing formal proofs, often by providing powerful proof automation. State-of-the-art proof assistants have proven successful in verification of software and hardware systems [69], but their potential use in the context of CMAS is yet to be uncovered. The major strength of a proof assistant is that every proof can be trusted as it is checked by the proof assistant's kernel. Exactly which proof assistant someone prefers is highly subjective. In this paper our work is carried out in the Isabelle/HOL proof assistant [58].

In the present paper, we formalize existing work on a verification framework for agents programmed in the language GOAL [22, 36, 37, 39]. The work we present is an extension of previous work [41, 43, 44, 47]. Section 5.2 elaborates on the contributions of this previous work.

The contribution of this paper is threefold:

- we present and describe how a temporal logic can be used to prove properties of agents specified in the agent logic,
- we demonstrate how this is achieved in practice by proving a correctness property for a simple example program, and finally
- we improve upon our previous work with a focus on concisely delivering the main results of the formalization in a format more friendly to those unfamiliar with proof assistants.

The paper is structured as follows. Section 5.2 places this work into context with related work. Section 5.3 provides an overview of the structure of the formalization. Section 5.4 describes the GOAL language and its formalization. Section 5.5 describes how to specify agents in a Hoare logic and how this translates to the low-level semantics. Section 5.6 introduces a temporal logic on top of the agent logic and describes how this can be used to prove temporal properties of agents. Section 5.7 demonstrates the use of the verification framework by means of a simple example program. Finally, Section 5.8 discusses limitations and further work and concludes on the paper.

This paper contains a number of Isabelle/HOL listings from the formalization source files referenced below, all of which have been verified. In a few cases, their presentation in this paper has been altered in minor ways for the sake of readability. For absolute precision, please refer to the full source which is publicly available online:

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

5.2 Related Work

The ability to demonstrate reliability of agent systems and autonomous systems has emerged as an important topic, perhaps sparked by the reemergence of a broader interest in AI. The following quote from a recent seminar establishes this point [28]:

> The aim of this seminar was to bring together researchers from various scientific disciplines, such as software engineering of autonomous systems, software verification, and relevant subareas of AI, such as ethics and machine learning, to discuss the emerging topic of the reliability of (multi-)agent systems and autonomous systems in particular. The ultimate aim of the seminar was to establish a new research agenda for engineering reliable autonomous systems.

The work presented in this paper embraces this perspective as we apply software verification techniques using the Isabelle/HOL proof assistant to verify agents programmed in the language GOAL.

The main approaches to agent verification can be categorized as testing, model checking and formal verification. Our approach falls into the latter category.

Testing is in general more approachable as it does not necessarily require advanced knowledge of verification systems. The main drawback is that it is non-exhaustive as tests explore a finite set of inputs and behaviors. In [54], the authors present a testing framework that automatically detects failures in cognitive agent programs. Failures are detected at run-time and are enabled by a specification of test conditions. While the work shows good promise, the authors note that more work needs to be dedicated to localizing failures.

Model checking is a technique where a system description is analyzed with respect to a property. It is then checked that all possible executions of the system

satisfy this property. Model checking is currently considered the state-of-the-art approach to an exhaustive demonstration of reliability for CMAS. In [23, 24], we are presented with a verification system for agent programming languages based on the belief-desire-intention model. In particular, an abstract layer maps the semantics of the programming languages to a model checking framework thus enabling their verification. The framework as a whole consists of two components: an Agent Infrastructure Layer (AIL) and an Agent Java Pathfinder (AJPF) which is a version of the Java Pathfinder (JPF) model checker [40]. The work contains a few example implementations. In particular, the semantics of the agent programming GOAL is mapped to AIL. While this remains purely speculative, it would be interesting to explore whether our formal verification approach could be generalized to an abstract agent infrastructure such as AIL.

Formal verification is a technique where properties are proved with respect to a formal specification using methods of formal mathematics. Because of the amount of work involved in conducting formal proofs and their susceptibility to human errors, such techniques are often applied with a theorem proving software tool such as a proof assistant. If the proof assistant is trusted, this in turn eliminates human errors in proofs while automation greatly reduces the amount of work required. The main drawback is the expert knowledge required to work with such software systems. In [31], the authors verify correctness for a class of algorithms which provides consistency guarantees for shared data in distributed systems. [69] provides a survey of the literature for engineering formally verified software. We have not been able to find related work that combines verification of agents with the use of proof assistants. We hope that the work presented in this paper can contribute to breaking the ice and stimulating interest in the potential of proof assistants for verification of agents.

We finally relate the contributions of this paper to our previous work. In [43], we outlined how to mechanically transform GOAL program code into the agent logic for GOAL formalized in this paper. In the present paper, we have not pursued the idea of formalizing this transformation in Isabelle/HOL. The example we present in Section 5.7 is based on actual program code, but has been significantly simplified for the sake of readability to hide programming quirks that do not translate elegantly and because our focus is on the Isabelle/HOL formalization. In [47], we argued for the use of theorem proving to verify CMAS and presented early work towards formalizing the framework. In [44], we go into the details of our early work on the formalization. In particular, there is more attention on the intricate details of formalizing the mental states of agents. Due to the sheer complexity and volume of the formal aspects we need to cover in the present paper, we cannot here dedicate attention to every detail. In [41], the emphasis is on formalizing actions and their effect on the mental state, and how to reason about such actions. The present paper is a formalization of the verification framework in its entirety and covers the most important aspects.

5.3 Formalizing GOAL in Isabelle/HOL

The framework formalization is around 2400 lines of code in total (178 KB) and loads in a few seconds. The BW4T example formalization is around 1100 lines of code in total (95 KB) and loads in about a minute. We want to emphasize that the example has not been optimized. With proper optimization, we expect the loading time for the example to be just a few seconds as well. The loading times have been tested on a modern laptop.

The *Gvf_Logic* theory sets up a framework for propositional classical logic, including a definition of semantics and a sequent calculus proof system. The *Gvf_Mental_States* theory introduces the definition of mental states and a logic for reasoning about mental states which is defined as a special case of propositional logic. The *Gvf_Actions* theory introduces concepts of agents, actions, transitions and traces. The *Gvf_Hoare_Logic* theory introduces a Hoare logic for reasoning about the (non-)effects of actions on mental states. The *Gvf_Agent_Specification* theory sets up a framework for specifying an agent using Hoare triples and a Hoare triple proof system. The *Gvf_Temporal_Logic* theory introduces a temporal logic to reason about temporal properties of specified agents. Finally, the *Gvf_Example_BW4T* theory proves the correctness of a simple example agent program.

Figure 5.1 illustrates the high-level components of the formalization and how they interact.

5.4 Logic for Agents

In this section, we describe a formalization of the GOAL language.

From the *Gvf_Logic* theory, we import a number of key definitions and proofs:

- $'a\ \Phi_P$: Datatype for propositional logic formulas with an arbitrary type $'a$ of atoms.
- $\models_P$: Semantic consequence for $'a\ \Phi_P$.
- $\vdash_P$: Sequent calculus proof system for $'a\ \Phi_P$.

The meaning of $\Gamma \models_P \Delta$ is that the truth of all formulas in the list of formulas Γ implies that at least one formula is true in the list of formulas Δ.

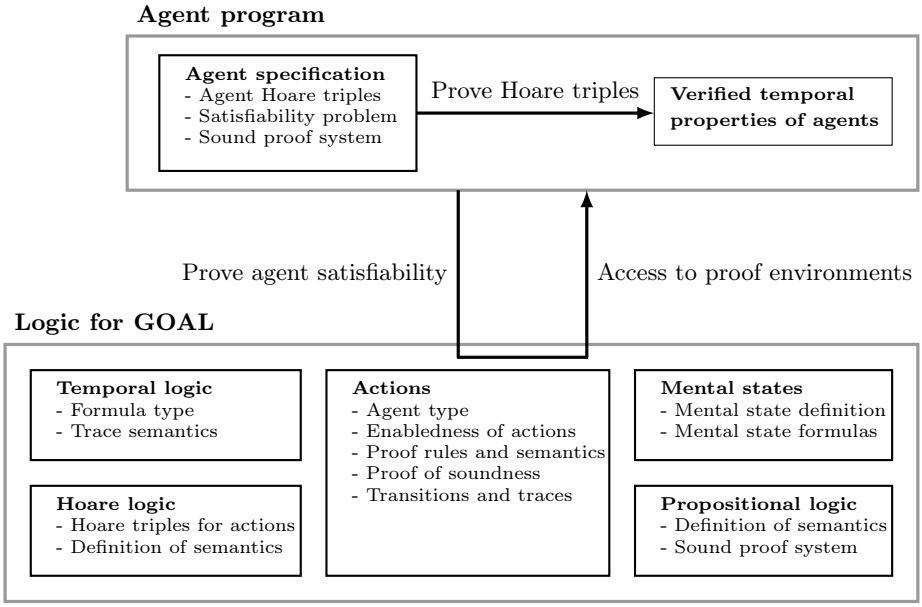

Figure 5.1: Visualization of the verification framework components.

The standard form of propositional logic, using string symbols for atoms, is denoted by the derived type Φ_L. Here, the semantics are defined over a model which maps string symbols to truth values.

Theorem 5.4.1 (Soundness of $\vdash_P$)**.** *The proof system $\vdash_P$ is sound with respect to the semantics of propositional logic formulas $\models_P$:*

$$\Gamma \vdash_P \Delta \implies \Gamma \models_P \Delta$$

Proof. By induction on the proof rules of $\vdash_P$. □

5.4.1 Mental States of Agents

We are now interested in formalizing a state-based semantics for GOAL. The state of an agent program is determined by the mental state of the agent. A mental state is modeled as two lists of formulas: a belief base and a goal base. The belief base formulas describe what the agents believe to be true while the goal base formulas describe what the agents want to achieve.

type-synonym *mental-state* = (Φ_L *list* × Φ_L *list*)

The original formulation describes these as sets, but we have chosen to work with lists. There are advantages and disadvantages to either approach.

Usually, we distinguish between the beliefs of the agent and the static knowledge about its world. In this setup, static knowledge is incorporated into the beliefs.

Due to limitations of the simple type system, we need a definition to formulate the special properties of a proper mental state:

definition *is-mental-state* :: *mental-state* ⇒ *bool* (∇) **where**
$\nabla\ M \equiv$ *let* (Σ, Γ) = M *in* $\neg\ \Sigma \models_P \bot \wedge (\forall \gamma \in set\ \Gamma.\ \neg\ \Sigma \models_P \gamma \wedge \neg \models_P (\neg\ \gamma))$

The above states that the belief base should be consistent, that no goal should be entailed by the belief base and that goals should be achievable.

The special belief and goal modalities, B and G respectively, enable the agent's introspective properties as defined by their semantics:

(Σ, -) $\models_M\ B\ \Phi = (\Sigma \models_P \Phi)$
(Σ, Γ) $\models_M\ G\ \Phi = (\neg\ \Sigma \models_P \Phi \wedge (\exists \gamma \in set\ \Gamma.\ \models_P (\gamma \longrightarrow \Phi)))$

The language of mental state formulas Φ_M can be perceived of as a special form of propositional logic where an atomic formula is either a belief or goal modality. This approach allows us to exploit the type variable $'a$ for atoms where the model is defined via the semantics of the belief and goal modalities.

We extend the proof system for propositional logic $\vdash_P$ with additional rules and axioms for the belief and goal modalities given in Table 5.1 to obtain a proof system for mental state formulas $\vdash_M$.

R_M-B	$\models_P \Phi \Longrightarrow \vdash_M B\ \Phi$
$A1_M$	$\vdash_M B\ (\Phi \longrightarrow \psi) \longrightarrow B\ \Phi \longrightarrow B\ \psi$
$A2_M$	$\vdash_M \neg\ (B\ \bot)$
$A3_M$	$\vdash_M \neg\ (G\ \bot)$
$A4_M$	$\vdash_M B\ \Phi \longrightarrow \neg\ (G\ \Phi)$
$A5_M$	$\models_P (\Phi \longrightarrow \psi) \Longrightarrow \vdash_M \neg\ (B\ \psi) \longrightarrow G\ \Phi \longrightarrow G\ \psi$

Table 5.1: Properties of beliefs and goals.

Theorem 5.4.2 (Soundness of $\vdash_M$)**.** *The proof system $\vdash_M$ is sound with respect to the semantics of mental state formulas $\models_M$ for any proper mental state (*∇ M*):*

$$\nabla\ M \Longrightarrow \vdash_M \Phi \Longrightarrow M \models_M \Phi$$

Proof. By induction on the proof rules and using Theorem 5.4.1. □

Notice that we assume $\nabla\ M$ which is necessary for the proof rules to hold semantically. Furthermore, in the soundness theorem for $\vdash_M$, there are no premises on the left-hand side. This is deliberate, as we want to match the formal notation for the semantics of mental states that takes on the left-hand side a mental state, i.e. $M \models_M \Phi$. We can solve this problem by incorporating premises in the target formula Φ by the use of implication, i.e. for premises $p_1,\ p_2,\ \ldots$ we have for the semantics $M \models_M p_1 \longrightarrow p_2 \longrightarrow \ldots \longrightarrow \Phi$.

5.4.2 Selecting and Executing Actions

In order to formalize a state-based semantics for GOAL, we need a notion of transitions between mental states of agents. We assume that agents themselves are the only ones capable of changing their mental state. As such, we can model transitions where an action executed in a given mental state has a predetermined outcome.

The low-level actions of GOAL are the basic actions. A basic action may be one of the two GOAL built-in actions for directly manipulating the goal base, or a user-defined action which is specific to the agent program:

datatype $cap = basic\ Bcap \mid adopt\ \Phi_L \mid drop\ \Phi_L$

The type *Bcap* is some type for identifying actions, in our case the type for strings.

An action transforms the mental state in which it is executed. In fact, for any action different from *adopt* and *drop* it is sufficient to consider only a transformation of the belief base as the effect on the goal base can be derived from the semantics of GOAL.

The transformation of mental states is partially defined by a function $\mathcal{T}$ of the type *Bcap* $\Rightarrow$ *mental-state* $\Rightarrow$ Φ_L *list option*: given a mental state and an action identifier, optionally a new belief base is returned. That $\mathcal{T}$ only optionally returns a new belief base is a result of the fact that a basic action may not be enabled in a given state. Notice that $\mathcal{T}$ is only concerned with the agent specific actions, namely those with an effect on the belief base.

The full effect on the mental state of executing a basic action is captured by a function $\mathcal{M}$:

fun *mst* :: *cap* $\Rightarrow$ *mental-state* $\Rightarrow$ *mental-state option* ($\mathcal{M}$) **where**
$\mathcal{M}$ (*basic n*) (Σ, Γ) = (*case* $\mathcal{T}$ *n* (Σ, Γ) *of*
  Some Σ' $\Rightarrow$ *Some* (Σ', [ψ<−Γ. ¬ Σ' $\models_P$ ψ])
  | - $\Rightarrow$ *None*)
$\mathcal{M}$ (*drop* Φ) (Σ, Γ) = *Some* (Σ, [ψ<−Γ. ¬ [ψ] $\models_P$ Φ])
$\mathcal{M}$ (*adopt* Φ) (Σ, Γ) = (*if* ¬ $\models_P$ (¬ Φ) ∧ ¬ Σ $\models_P$ Φ *then*
  Some (Σ, *List.insert* Φ Γ) *else None*)

The case for agent specific actions captures the default commitment strategy where goals are only removed once achieved.

On top of basic actions we introduce the notion of a conditional action. A conditional action consists of a basic action and a condition φ, denoted $\varphi \triangleright$ *do a*. This condition is specified as a mental state formula which is evaluated in a given mental state. The action can only be executed if the condition is met. As such, the enabledness of a conditional action depends on both $\mathcal{T}$ and φ. The set of conditional actions for an agent is denoted Π.

Combining the notion of conditional actions and a mental state transformer gives rise to the concept of a transition between two states due to the execution of an action, captured by the following definition:

definition *transition* :: *mental-state* $\Rightarrow$ *cond-act* $\Rightarrow$ *mental-state* $\Rightarrow$ *bool* (- →- -) **where**
  $M \rightarrow b\ M' \equiv b \in \Pi \wedge M \models_M (fst\ b) \wedge \mathcal{M}\ (snd\ b)\ M = Some\ M'$

In the formalization, a conditional action b is a tuple and *fst b* gives its condition while *snd b* gives its basic action. The definition states that a transition $M \rightarrow (\varphi \triangleright do\ a)\ M'$ exists if $\varphi \triangleright do\ a$ is in Π, the condition φ holds in M and M' is the resulting state.

A run of an agent program can be understood as a sequence of mental states interleaved with conditional actions (M_0, b_0, M_1, b_1, …, M_i, b_i, M_{i+1}, …).

We do not consider the use of a stop criterion and instead assume the program to continue indefinitely. It is not a criterion for scheduled actions (those in the trace) to necessarily be enabled and thus executed. In such cases, we have $M_i = M_{i+1}$. We call a possible run of the agent a trace. The *codatatype* command allows for a coinductive datatype:

codatatype *trace* = *Trace mental-state cond-act* × *trace*

We further define functions *st-nth* and *act-nth* which give the i-th state and conditional action of a trace, respectively.

Analogous to mental states, our definition of a trace includes all elements of the simple type, so we need a definition:

definition *is-trace* :: *trace* ⇒ *bool* **where**
is-trace s ≡ ∀ *i*. (*act-nth s i*) ∈ Π ∧
(((*st-nth s i*) →(*act-nth s i*) (*st-nth s* (*i*+*1*))) ∨
¬(∃ *M*. (*st-nth s i*) →(*act-nth s i*) *M*) ∧
(*st-nth s i*) = (*st-nth s* (*i*+*1*)))

The definition requires that for all M_i, b_i, M_{i+1} either a transition $M_i \rightarrow b_i\ M_{i+1}$ exists or the action is not enabled and $M_i = M_{i+1}$. The definition makes it explicit that if a transition exists from M_i then it is to M_{i+1}.

Figure 5.2 illustrates the set of all traces and highlights a single trace.

For any scheduling of actions by the agent, we assume *weak fairness*. Consequently, we have that the traces are *fair*:

definition *fair-trace* :: *trace* ⇒ *bool* **where**
fair-trace s ≡ ∀ *b* ∈ Π. ∀ *i*. ∃ *j* > *i*. *act-nth s j* = *b*

At any point in a fair trace, for any action there always exists a future point where it is scheduled for execution. The meaning of an agent is defined as the set of all possible fair traces starting from a predetermined initial state M_0:

definition *Agent* :: *trace set* **where**
Agent ≡ {*s* . *is-trace s* ∧ *fair-trace s* ∧ *st-nth s 0* = M_0}

With the newly introduced terminology, it becomes rather straightforward to express the semantics of the new components in formulas regarding enabledness:

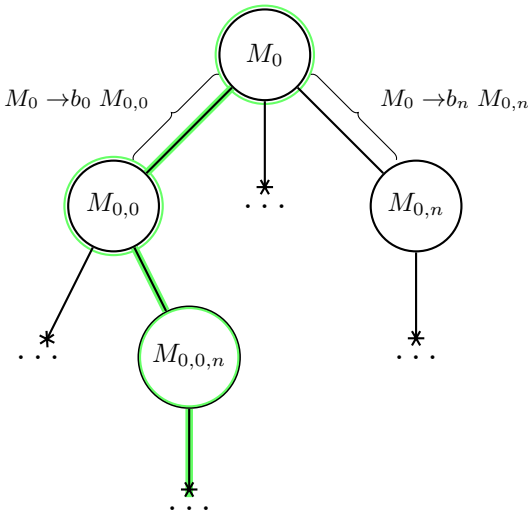

Figure 5.2: Visualization of the set of all traces, highlighting a single trace.

$M \models_E (\textit{enabled-basic}\ a) = (a \in \textit{Cap} \wedge \mathcal{M}\ a\ M \neq \textit{None})$
$M \models_E (\textit{enabledond}\ b) = (\exists\, M'.\ (M \to b\ M'))$

Here, *Cap* is the set of the agent's basic actions. For the second case, a similar check is built-in to the transition definition.

We introduce the shorthand notation $\varphi[s\ i]$ for evaluating the mental state formula φ in the i-th state of a trace s.

An extended proof system $\vdash_E$ is obtained by including the rules of Table 5.2 which state syntactic properties of enabledness.

$E1_E$	$(\varphi \triangleright \textit{do}\ a) \in \Pi \Longrightarrow \vdash_E \textit{enabled}\ (\varphi \triangleright \textit{do}\ a) \longleftrightarrow (\varphi \wedge \textit{enabledb}\ a)$
$E2_E$	$\vdash_E \textit{enabledb}\ (\textit{drop}\ \Phi)$
$R3_E$	$\neg \models_P (\neg\ \Phi) \Longrightarrow \vdash_E \neg\ (B_E\ \Phi) \longleftrightarrow \textit{enabledb}\ (\textit{adopt}\ \Phi)$
$R4_E$	$\models_P (\neg\ \Phi) \Longrightarrow \vdash_E \neg\ (\textit{enabledb}\ (\textit{adopt}\ \Phi))$
$R5_E$	$\forall\, M.\ \nabla M \longrightarrow \mathcal{T}\ a\ M \neq \textit{None} \Longrightarrow \vdash_E \textit{enabledb}\ (\textit{basic}\ a)$

Table 5.2: Enabledness of actions.

Theorem 5.4.3 (Soundness of $\vdash_E$)**.** *The proof system $\vdash_E$ is sound with respect to the semantics of mental state formulas including enabledness $\models_E$ for any proper mental state (∇ M):*

$\nabla\ M \Longrightarrow \vdash_E \varphi \Longrightarrow M \models_E \varphi$

Proof. By induction on the proof rules and using Theorem 5.4.2. □

5.4.3 Hoare Logic for GOAL

To reason about transition steps due to execution of actions, we set up a specially tailored Hoare logic. By means of Hoare triples, we can specify the effect of executing an action using mental state formulas as pre- and postconditions. We distinguish between Hoare triples for basic actions and conditional actions, but as becomes apparent later, there is a close relationship between the two. We introduce the usual Hoare triple notation: $\{\varphi\}\ a\ \{\psi\}$ for basic actions and $\{\varphi\}\ \upsilon \triangleright do\ b\ \{\psi\}$ for conditional actions. The notation masks two constructors for a simple datatype *hoare-triple* consisting of two formulas and a basic or conditional action.

We now give the semantics of Hoare triples. The pre- and postconditions can only be evaluated given a current mental state. However, for the definition of the semantics we quantify over all mental states. In other words, a Hoare triple is only true if it holds at any point in the agent program:

$$\begin{aligned}
&\models_H \{\ \varphi\ \}\ a\ \{\ \psi\ \} = (\forall\, M.\\
&\quad (M \models_E \varphi \wedge (enabledb\ a) \longrightarrow the\ (\mathcal{M}\ a\ M) \models_M \psi) \wedge\\
&\quad (M \models_E \varphi \wedge \neg(enabledb\ a) \longrightarrow M \models_M \psi))\\
&\models_H \{\ \varphi\ \}\ (\upsilon \triangleright do\ b)\ \{\ \psi\ \} = (\forall\ s \in Agent.\ \forall\, i.\\
&\quad (\varphi[s\ i] \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ i) \longrightarrow \psi[s\ (i{+}1)])
\end{aligned}$$

The first case is for basic actions. In any state, if the precondition is true, the postcondition should hold in the state obtained by executing the action a. Otherwise, it should hold in the same state as the action is not enabled and thus not executed. The second case is for conditional actions. The case is analogous, but the enabledness of the action, and thus whether the successor state is unchanged, is implicit due to the definition of traces.

Lemma 5.4.4 (Relation between Hoare triples for basic and conditional actions)**.**

$$\begin{aligned}
&\models_H \{\ \varphi \wedge \psi\ \}\ a\ \{\ \varphi'\ \} \Longrightarrow \forall\ s \in Agent.\ \forall\, i.\ ((\varphi \wedge \neg\psi) \longrightarrow \varphi')[s\ i] \Longrightarrow\\
&\quad \models_H \{\ \varphi\ \}\ (\psi \triangleright do\ a)\ \{\ \varphi'\ \}
\end{aligned}$$

Proof. We need to show $\varphi'[s\ (i{+}1)]$. We can assume $(\varphi[s\ i] \wedge (v \triangleright do\ b) = (act\text{-}nth\ s\ i)$. We then distinguish between the cases of whether $\psi[s\ i]$ holds. If it holds, the basic action Hoare triple can be used to infer $\varphi'[s\ (i{+}1)]$. If it does not hold, the result follows from the second assumption as the action a is not enabled and thus *st-nth s i = st-nth (i+1).* □

Lemma 5.4.4 is used to prove Theorem 5.5.3.

5.5 Specification of Agents

In this section, we are concerned with the specification of agent programs using Hoare triples and finding a Hoare system to be able to derive Hoare triples for the specified agent program. Because the low-level semantics of GOAL is not based on Hoare logic, we need to show that the low-level semantics can be derived from an agent specification based on Hoare triples. More specifically, it suffices to show that it is possible to come up with a definition of $\mathcal{T}$ which satisfies all the specified Hoare triples.

We define an agent specification as a list:

type-synonym *ht-specification = ht-spec-elem list*

The elements of such a list are tuples containing an action identifier, a mental state formula which is the decision rule for the action and lastly a list of Hoare triples which specify the frame and effect axioms for the action:

type-synonym *ht-spec-elem = Bcap* $\times$ Φ_M $\times$ *hoare-triple list*

We have not touched upon this earlier, but frame axioms specify what does not change when executing an action and effect axioms specify what does change, i.e. what are the effects of executing the action. Another note is that we only allow a single decision rule in our setup. We could allow for multiple rules per action. Instead, we assume that if there are multiple rules they have been condensed to a single rule using disjunction.

5.5.1 Satisfiability Problem

Our task is now to show that a low-level semantics can be derived from a specification, i.e. that there exists some $\mathcal{T}$ which complies with the specification. This may also be seen as a model existence problem which has already been studied for other areas of logic. Another perspective on this problem is to consider it as a satisfiability problem, or equivalently that there are no contradictions.

The definition of satisfiability is split into two parts, quantifying over any proper mental state and every element of the specification:

definition *satisfiable* :: *ht-specification* ⇒ *bool* **where**
  satisfiable $S \equiv \forall M.\ \nabla\ M \longrightarrow (\forall s \in set\ S.\ sat\text{-}l\ M\ s \wedge sat\text{-}r\ M\ s)$

The first part is concerned with those mental states where the action in question is enabled:

fun *sat-l* :: *mental-state* ⇒ *ht-spec-elem* ⇒ *bool* **where**
 sat-l $M\ (a, \Phi, hts) = (M \models_M \Phi \longrightarrow (\exists \Sigma.\ sat\text{-}b\ M\ hts\ \Sigma))$

When the action is enabled, for all states there should exist a belief base which satisfies all Hoare triples for this action:

definition *sat-b* :: *mental-state* ⇒ *hoare-triple list* ⇒ Φ_L *list* ⇒ *bool* **where**
  sat-b $M\ hts\ \Sigma \equiv$
    $(\neg\ fst\ M \models_P \bot \longrightarrow \neg\ \Sigma \models_P \bot) \wedge$
    $(\forall ht \in set\ hts.\ M \models_M pre\ ht \longrightarrow (\Sigma, [\psi <- snd\ M.\ \neg\ \Sigma \models_P \psi]\ ) \models_M post\ ht)$

For a belief base to be satisfiable in this context, we require:

1. that the consistency is preserved, and
2. that the postcondition holds in the new mental state if the precondition holds in the current mental state (the new goal base is derived by removing achieved goals from the current goal base).

The second part of the satisfiability definition is concerned with those mental states where the action in question is not enabled:

fun *sat-r* :: *mental-state* ⇒ *ht-spec-elem* ⇒ *bool* **where**
sat-r M (-, Φ, *hts*) =
(*M* ⊨$_M$ ¬ Φ ⟶ (∀ *ht* ∈ *set hts*. *M* ⊨$_M$ *pre ht* ⟶ *M* ⊨$_M$ *post ht*))

In such a case the postcondition should hold in the same state if the precondition holds.

Because of the way we have set up the specification type, a definition *is-ht-specification* ensures that the specification is satisfiable and that each element reflects the specification of a distinct action.

We briefly mentioned that satisfiability entails the existence of a $\mathcal{T}$ which matches the agent specification. To show this, we first define what it means for a $\mathcal{T}$ to comply with an agent specification:

definition *complies* :: *ht-specification* ⇒ *bel-upd-t* ⇒ *bool* **where**
complies S $\mathcal{T}$ ≡ (∀ *s*∈*set S*. *complies-hts s* $\mathcal{T}$) ∧
(∀ *n*. *n* ∉ *set* (*map fst S*) ⟶ (∀ *M*. $\mathcal{T}$ *n M* = *None*))

Again we can define this for each element of specification individually. Furthermore, we require that $\mathcal{T}$ is only defined for action identifiers present in the specification.

definition *complies-hts* :: (*Bcap* × Φ$_M$ × *hoare-triple list*) ⇒ *bel-upd-t* ⇒ *bool* **where**
complies-hts s $\mathcal{T}$ ≡ ∀ *ht*∈*set* (*snd* (*snd s*)). *is-htb-basic ht* ∧
(∀ *M*. ∇*M* ⟶ *complies-ht M* $\mathcal{T}$ (*fst* (*snd s*)) (*the* (*htb-basic-unpack ht*)))

The following definition has a convoluted syntax. Essentially, we quantify over all proper mental states and assert compliance for each Hoare triple:

fun *complies-ht* :: *mental-state* ⇒ *bel-upd-t* ⇒ Φ$_M$ ⇒ (Φ$_M$ × *Bcap* × Φ$_M$) ⇒ *bool* **where**
complies-ht M $\mathcal{T}$ Φ (φ, *n*, ψ) =
((*M* ⊨$_M$ Φ ⟷ $\mathcal{T}$ *n M* ≠ *None*) ∧
(¬ (*fst M*) ⊨$_P$ ⊥ ⟶ $\mathcal{T}$ *n M* ≠ *None* ⟶ ¬*the* ($\mathcal{T}$ *n M*) ⊨$_P$ ⊥) ∧
(*M* ⊨$_M$ φ ∧ *M* ⊨$_M$ Φ ⟶ *the* ($\mathcal{M}$∗ $\mathcal{T}$ (*basic n*) *M*) ⊨$_M$ ψ) ∧
(*M* ⊨$_M$ φ ∧ *M* ⊨$_M$ ¬ Φ ⟶ *M* ⊨$_M$ ψ))

Note that in the preceding definition, $\mathcal{M}$∗ $\mathcal{T}$ corresponds to the mental state transformer $\mathcal{M}$ where $\mathcal{T}$ is not fixed by the agent. Furthermore, the definition packs a number of important properties:

1. that the formula φ is the sole factor for enabledness of the action,
2. that consistency is preserved for belief bases, and lastly
3. that it matches the semantics of the Hoare triple.

Arguably, the definitions of satisfiability and compliance could be optimized for readability and less redundancy.

We now show that compliance is entailed by satisfiability. An interesting partial result is that because each element of specification is for a distinct action, the existence of a (partial) $\mathcal{T}$ can be shown for each action and used to show the existence of a $\mathcal{T}$ for the full specification:

Lemma 5.5.1 (Disjoint compliance)**.** *The existence of a $\mathcal{T}$ for each element of the specification can be used to show the existence of a $\mathcal{T}$ for the full specification.*

$$\textit{is-ht-specification } S \Longrightarrow \forall\, s{\in}\textit{set } S.\ \exists \mathcal{T}.\ \textit{complies-hts } s\ \mathcal{T} \Longrightarrow \exists \mathcal{T}.\ \textit{complies } S\ \mathcal{T}$$

Proof. Since there is no overlap between each partial $\mathcal{T}_x$ obtained from the assumptions, we combine the partial functions into a single function $\mathcal{T}$ which complies with the specification by construction. □

We then prove that for any specification which follows the definition (most importantly, it is thus satisfiable) there exists a compliant $\mathcal{T}$.

Lemma 5.5.2 (Compliance)**.** *There exists a belief update function $\mathcal{T}$ for any proper agent specification which is satisfiable.*

$$\textit{is-ht-specification } S \Longrightarrow \exists \mathcal{T}.\ \textit{complies } S\ \mathcal{T}$$

Proof. By construction of a $\mathcal{T}$ for each specification element using the definition of satisfiable bases and ultimately using Lemma 5.5.1. □

5.5.2 Derived Proof System

Our primary focus is on proving the truth of particular Hoare triples. Exactly how this plays into proving properties of agents becomes apparent later. To this end, we need a Hoare logic proof system. While we can state the general proof rules for Hoare triples, as well as axioms regarding pre- and postconditions

involving the special belief and goal modalities, the most important properties of any agent program depend on the agent specification. The user needs to specify a number of effect and frame axioms. They state what is changed and what is not changed by executing the action. Outside of the general rules for proving Hoare triples, we include a special import rule which allows for any of the specified Hoare triples as axioms. The soundness of the import rule follows directly from the compliance of $\mathcal{T}$.

The proof system is defined inductively:

inductive $derive_H :: hoare\text{-}triple \Rightarrow bool\ (\vdash_H)$

The *import* rule allows for a specified Hoare triple as an axiom:

$$(n, \Phi, s) \in set\ S \Longrightarrow \{\ \varphi\ \}\ (basic\ n)\ \{\ \psi\ \} \in set\ s \Longrightarrow \vdash_H \{\ \varphi\ \}\ (basic\ n)\ \{\ \psi\ \}$$

The *persist* rule states that a goal persists as either a belief or goal unless it is dropped:

$$\neg\ is\text{-}drop\ a \Longrightarrow \vdash_H \{\ G\ \Phi\ \}\ a\ \{\ B\ \Phi \vee G\ \Phi\ \}$$

The *inf* rule states that nothing happens if the precondition implies that the action is not enabled:

$$\models_E (\varphi \longrightarrow \neg(enabledb\ a)) \Longrightarrow \vdash_H \{\ \varphi\ \}\ a\ \{\ \varphi\ \}$$

The following rules state important properties of the *adopt* and *drop* actions:

adoptB:	$\vdash_H \{\ B\ \Phi\ \}\ (adopt\ \psi)\ \{\ B\ \Phi\ \}$
adoptNegB:	$\vdash_H \{\ \neg\ (B\ \Phi)\ \}\ (adopt\ \psi)\ \{\ \neg\ (B\ \Phi)\ \}$
dropB:	$\vdash_H \{\ B\ \Phi\ \}\ (drop\ \psi)\ \{\ B\ \Phi\ \}$
dropNegB	$\vdash_H \{\ \neg\ (B\ \Phi)\ \}\ (drop\ \psi)\ \{\ \neg\ (B\ \Phi)\ \}$
adoptBG:	$\neg \models_P (\neg\ \Phi) \Longrightarrow \vdash_H \{\ \neg\ (B\ \Phi)\ \}\ (adopt\ \Phi)\ \{\ G\ \Phi\ \}$
adoptG:	$\vdash_H \{\ G\ \Phi\ \}\ (adopt\ \psi)\ \{\ G\ \Phi\ \}$ |
adoptNegG:	$\neg \models_P (\psi \longrightarrow \Phi) \Longrightarrow \vdash_H \{\ \neg\ (G\ \Phi)\ \}\ (adopt\ \psi)\ \{\ \neg\ (G\ \Phi)\ \}$
dropG:	$\models_P (\Phi \longrightarrow \psi) \Longrightarrow \vdash_H \{\ G\ \Phi\ \}\ (drop\ \psi)\ \{\ \neg\ (G\ \Phi)\ \}$
dropNegG:	$\vdash_H \{\ \neg(G\ \Phi)\ \}\ (drop\ \psi)\ \{\ \neg\ (G\ \Phi)\ \}$
dropGCon:	$\vdash_H \{\ \neg(G\ (\Phi \wedge \psi)) \wedge (G\ \Phi)\ \}\ (drop\ \psi)\ \{\ G\ \Phi\ \}$

Finally, we have the structural rules:

$$\begin{array}{ll}
\textit{rCondAct:} & \vdash_H \{\ \varphi \wedge \psi\ \}\ a\ \{\ \varphi'\ \} \Longrightarrow \models_M (\varphi \wedge \neg\psi \longrightarrow \varphi') \\
& \quad \Longrightarrow \vdash_H \{\ \varphi\ \}\ (\psi \triangleright \mathit{do}\ a)\ \{\ \varphi'\ \} \\
\textit{rImp:} & \models_M (\varphi' \longrightarrow \varphi) \Longrightarrow \vdash_H \{\ \varphi\ \}\ a\ \{\ \psi\ \} \Longrightarrow \models_M (\psi \longrightarrow \psi') \Longrightarrow \\
& \quad \vdash_H \{\ \varphi'\ \}\ a\ \{\ \psi'\ \} \\
\textit{rCon:} & \vdash_H \{\ \varphi_1\ \}\ a\ \{\ \psi_1\ \} \Longrightarrow \vdash_H \{\ \varphi_2\ \}\ a\ \{\ \psi_2\ \} \Longrightarrow \\
& \quad \vdash_H \{\ \varphi_1 \wedge \varphi_2\ \}\ a\ \{\ \psi_1 \wedge \psi_2\ \} \\
\textit{rDis:} & \vdash_H \{\ \varphi_1\ \}\ a\ \{\ \psi\ \} \Longrightarrow \vdash_H \{\ \varphi_2\ \}\ a\ \{\ \psi\ \} \Longrightarrow \\
& \quad \vdash_H \{\ \varphi_1 \vee \varphi_2\ \}\ a\ \{\ \psi\ \}
\end{array}$$

Due to the sheer number of proof rules, proving the soundness of the system becomes quite involved.

Theorem 5.5.3 (Soundness of $\vdash_H$)**.** *The Hoare system $\vdash_H$ is sound with respect to the semantics of Hoare triples $\models_H$.*

$$\vdash_H H \Longrightarrow \models_H H$$

Proof. By induction on the proof rules. The import rule follows directly from Lemma 5.5.2. The rule for conditional actions follows from Lemma 5.4.4. The remaining rules follow from the semantics of mental states $\models_E$ and the semantics of Hoare triples $\models_H$. □

5.6 Proving Correctness

In this section, we describe how to prove temporal properties of agents by means of a temporal logic which is constructed on top of the logic for GOAL. Ultimately, we show that proofs of certain liveness and safety properties can be reduced to proofs of Hoare triples in the Hoare logic.

5.6.1 Temporal Logic

We start by setting up a datatype for temporal logic formulas with just two temporal operators and where the type variable $'a$ allows for any type of atoms:

datatype $'a\ \Phi_T =$
  $F\ (\bot_T)\ |$

Atom $'a$ |
Negation $('a\ \Phi_T)\ (\neg_T)$ |
Implication $('a\ \Phi_T)\ ('a\ \Phi_T)$ (**infixr** $\longrightarrow_T$ *60*) |
Disjunction $('a\ \Phi_T)\ ('a\ \Phi_T)$ (**infixl** $\vee_T$ *70*) |
Conjunction $('a\ \Phi_T)\ ('a\ \Phi_T)$ (**infixl** $\wedge_T$ *80*) |
init |
until $('a\ \Phi_T)\ ('a\ \Phi_T)$

The Boolean operators each have a subscript to avoid ambiguity, e.g. $\longrightarrow_T$ for implication.

Additional temporal operators can be defined by combining the existing ones. The *always* operator $\Box$ states that the operand remains true forever:

definition *always* :: $'a\ \Phi_T \Rightarrow 'a\ \Phi_T$ ($\Box$) **where**
$\Box\varphi \equiv \varphi$ *until* $\bot_T$

The *eventuality* operator $\diamond$ states that the operand is true at some point:

definition *eventuality* :: $'a\ \Phi_T \Rightarrow 'a\ \Phi_T$ ($\diamond$) **where**
$\diamond\varphi \equiv \neg_T\ (\Box\ (\neg_T\varphi))$

The *unless* operator states for φ *unless* ψ that if φ becomes true then it remains true until ψ becomes true:

definition *unless* :: $'a\ \Phi_T \Rightarrow 'a\ \Phi_T \Rightarrow 'a\ \Phi_T$ **where**
φ *unless* $\psi \equiv \varphi \longrightarrow_T (\varphi$ *until* $\psi)$

In the following, the type Φ_{TM} is for temporal logic with the belief and goal modalities as atoms. The semantics of temporal logic for agents is evaluated in terms of a trace s and a natural number i, indicating that the formula is to be evaluated in the i-th state of trace s:

$s,\ i \models_T \bot_T = False$
$s,\ i \models_T (Atom\ x) = ((Atom\ x)[s\ i]_M)$
$s,\ i \models_T (\neg_T\ p) = (\neg\ (s,\ i \models_T p))$
$s,\ i \models_T (p \longrightarrow_T q) = ((s,\ i \models_T p) \longrightarrow (s,\ i \models_T q))$
$s,\ i \models_T (p \vee_T q) = ((s,\ i \models_T p) \vee (s,\ i \models_T q))$
$s,\ i \models_T (p \wedge_T q) = ((s,\ i \models_T p) \wedge (s,\ i \models_T q))$
$s,\ i \models_T init = (i = 0)$
$s,\ i \models_T (\varphi\ until\ \psi) = ((\exists\ j \geq i.\ s,\ j \models_T \psi \wedge (\forall k \geq i.\ j > k \longrightarrow s,\ k \models_T \varphi)) \vee (\forall k \geq i.\ s,\ k \models_T \varphi))$

The Boolean operators are defined using Isabelle's operators. Identical results can be achieved using the more traditional if-then-else constructs if one desires a more programming-like style. The case for atoms simply delegates the task to the semantics functions for mental state formulas. The case for *init* is true when $i = 0$, i.e. when we are at the very first state of the trace. The more complicated case for *until* warrants further explanation: either φ remains true until a future point j where ψ becomes true, or φ remains true forever.

5.6.2 Liveness and Safety Properties

There is a close relationship between proving Hoare triples and proving liveness and safety properties of an agent. Concerning safety, we can show that φ is a stable property by proving φ *unless F*. In case we also have *init* $\longrightarrow\varphi$, we say that φ is an invariant of the agent program. We now prove that, due to the *unless* operator, this safety property can be reduced to proving a Hoare triple for each conditional action.

Theorem 5.6.1. *After executing any action from* Π *either* φ *persists or* ψ *becomes true and we can conclude* φ unless ψ *and conversely.*

$$\forall (\upsilon \triangleright do\ b) \in \Pi.\ \models_H \{\ \varphi \wedge \neg\ \psi\ \}\ (\upsilon \triangleright do\ b)\ \{\ \varphi \vee \psi\ \})$$
$$\longleftrightarrow\ \ Agent \models_T \varphi\ unless\ \psi$$

Proof. The $\longrightarrow$ direction is shown by contraposition where the semantics of temporal logic leads us to a contradiction. The $\longleftarrow$ direction is shown by a case distinction of $\models_M \varphi$ in some arbitrarily chosen state. □

Liveness properties involve showing that a particular state is always reached from a given situation. A certain subclass of these properties is captured by the *ensures* operator:

definition *ensures* :: $'a\ \Phi_T \Rightarrow 'a\ \Phi_T \Rightarrow 'a\ \Phi_T$ **where**
φ *ensures* $\psi \equiv (\varphi$ *unless* $\psi) \wedge_T (\varphi \longrightarrow_T \diamond\psi)$

Here, φ *ensures* ψ informally means that φ guarantees the realization of ψ.

Also for the *ensures* operator we can show that it can be reduced to proofs of Hoare triples.

Theorem 5.6.2. *The proof of an* ensures *property can be reduced to the proof of a set of Hoare triples.*

$$\forall\, b \in \Pi.\ \models_H \{\ \varphi \wedge \neg\ \psi\ \}\ b\ \{\ \varphi \vee \psi\ \} \Longrightarrow$$
$$\exists\, b \in \Pi.\ \models_H \{\ \varphi \wedge \neg\ \psi\ \}\ b\ \{\ \psi\ \} \Longrightarrow$$
$$\mathit{Agent} \models \varphi\ \mathit{ensures}\ \psi$$

Proof. From the fact that any trace in *Agent* is a fair trace, we obtain a contradiction using the semantics of temporal logic $\models_T$ and of Hoare triples $\models_H$. □

Finally, we introduce the temporal operator $\mapsto$ ("leads to"). The property $\varphi \mapsto \psi$ is similar to *ensures* except that it does not require φ to remain true until ψ is realized. It is defined from the ensures operator inductively:

inductive *leads-to* :: $\Phi_{TM} \Rightarrow \Phi_{TM} \Rightarrow$ *bool* (**infix** $\mapsto$ *55*) **where**
base: $\forall\, s \in \mathit{Agent}.\ \forall\, i.\ s,\ i \models_T (\varphi\ \mathit{ensures}\ \psi) \Longrightarrow \varphi \mapsto \psi$ |
imp: $\varphi \mapsto \chi \Longrightarrow \chi \mapsto \psi \Longrightarrow \varphi \mapsto \psi$ |
disj: $\forall\, \varphi \in \mathit{set}\ \varphi_L.\ \varphi \mapsto \psi \Longrightarrow \mathit{disL}\ \varphi_L \mapsto \psi$

The rule *imp* states a transitive property and *disj* states a disjunctive property. The function *disL* forms a disjunction from a list of formulas:

fun *disL*:: $'a\ \Phi_T\ \mathit{list} \Rightarrow 'a\ \Phi_T$ **where**
$\mathit{disL}\ [] = \bot_T$ |
$\mathit{disL}\ [\varphi] = \varphi$ |
$\mathit{disL}\ (\varphi\ \#\ \varphi_L) = \varphi \vee_T \mathit{disL}\ \varphi_L$

We can prove that the temporal operator $\mapsto$ can be used to state a correctness property for agents.

Lemma 5.6.3. *The proof of a certain class of temporal properties can be reduced to a proof of the "leads to" operator* $\mapsto$*.*

$$\varphi \mapsto \psi \Longrightarrow \forall\, s \in \mathit{Agent}.\ \forall\, i.\ s,\ i \models_T (\varphi \longrightarrow_T \Diamond\psi)$$

Proof. By induction on the rules and using the semantics of temporal logic $\models_T$. □

Lemma 5.6.3 is used to prove that temporal logic statements of the form $P \mapsto Q$ in fact state temporal properties of agents.

5.7 An Example Agent

In this section, we showcase how to use the verification framework in Isabelle/HOL to prove the correctness of a simple agent which can solve a simple task to collect a block. Informally, the agent initially is located in a special dropzone location where a block is to be delivered. The agent must go to a room containing such a block. Before the agent can pick up a block, it must move right next to it. Finally, the agent must return to the dropzone and deliver the block.

The belief base of the initial mental state is:

[*in-dropzone*, ¬ *in-room*, ¬ *holding*, ¬ *at-block*]

The goal base of the initial mental state is:

[*collect*]

The initial belief and goal bases capture the initial configuration:

- The agent is located in the dropzone.
- Naturally, the agent is therefore not in a room containing a block, and consequently also not next to a block.
- The agent is not holding a block.
- The goal of the agent is to collect a block.

Furthermore, our example agent has a number of available actions:

- *go-dropzone*: The agent moves to the dropzone.
- *go-room*: The agent moves to a room containing a block.
- *go-block*: When in a room containing a block, the agent moves right next to the block.
- *pick-up*: If the agent is right next to a block, the agent will pick it up.
- *put-down*: If the agent is carrying a block, the agent will put it down.

Each action has a condition which states when the action can be performed. This is specified by the following formulas for enabledness:

$$\mathit{enabled(go\text{-}dropzone)} \equiv B\ \mathit{in\text{-}room} \wedge B\ \mathit{holding}$$

$$\mathit{enabled(go\text{-}room)} \equiv B\ \mathit{in\text{-}dropzone} \wedge \neg\ (B\ \mathit{holding})$$

$$\mathit{enabled(go\text{-}block)} \equiv B\ \mathit{in\text{-}room} \wedge \neg\ (B\ \mathit{at\text{-}block}) \wedge \neg\ (B\ \mathit{holding})$$

$$\mathit{enabled(pick\text{-}up)} \equiv B\ \mathit{at\text{-}block} \wedge \neg\ (B\ \mathit{holding})$$

$$\mathit{enabled(put\text{-}down)} \equiv B\ \mathit{holding} \wedge B\ \mathit{in\text{-}dropzone}$$

Because of the simplicity of our example, only one of these conditions is true in any given state. In other words, there is only ever one meaningful action.

Furthermore, we must specify effect axioms for the actions. In our case, each action has exactly one effect axiom:

$$\{\ B\ \mathit{in\text{-}room} \wedge B\ \mathit{holding}\ \}\ \mathit{go\text{-}dropzone}\ \{\ B\ \mathit{in\text{-}dropzone}\ \}$$

$$\{\ B\ \mathit{in\text{-}dropzone} \wedge \neg\ (B\ \mathit{holding})\ \}\ \mathit{go\text{-}room}\ \{\ B\ \mathit{in\text{-}room}\ \}$$

$$\{\ B\ \mathit{in\text{-}room} \wedge \neg\ (B\ \mathit{at\text{-}block}) \wedge \neg\ (B\ \mathit{holding})\ \}\ \mathit{go\text{-}block}\ \{\ B\ \mathit{at\text{-}block}\ \}$$

$$\{\ B\ \mathit{at\text{-}block} \wedge \neg\ (B\ \mathit{holding})\ \}\ \mathit{pick\text{-}up}\ \{\ B\ \mathit{holding}\ \}$$

$$\{\ B\ \mathit{in\text{-}dropzone} \wedge B\ \mathit{holding}\ \}\ \mathit{put\text{-}down}\ \{\ B\ \mathit{collect}\ \}$$

Since we do not have the space to go through the details, the specification of frame axioms and proofs of invariants has been left out of the paper.

We define the belief update function as some $\mathcal{T}$ which complies with the specification. We are allowed to do this if we can prove that such a specification exists.

definition $\mathcal{T}_x \equiv \mathit{SOME}\ \mathcal{T}.\ \mathit{complies}\ S_x\ \mathcal{T}$

We need to prove that our different example components actually compose a single agent program:

interpretation *bw4t*: *single-agent-program* $\mathcal{T}_x$ *set* Π_x M_{0x} S_x

The requirements of a single agent program has not been described earlier. It requires showing the existence of some $\mathcal{T}$ which complies with the specification. This is due to the definition of $\mathcal{T}_x$ using *SOME* which is based on Hilbert's epsilon operator i.e. the axiom of choice. The main issue is thus to show the satisfiability of the specification due to Lemma 5.5.2. The proof is rather lengthy and is not shown here.

The main point of interest is the proof of the statement involving the "leads to" operator:

lemma ‹*B in-dropzone* ∧ ¬ (*B at-block*) ∧ ¬ (*B holding*) ∧ *G collect* ↦ *B collect*›

This captures that the desired state, where the agent believes to have collected the block, is reached from the initial configuration of the agent. The inductive definition of the operator makes it possible to split the proof into subproofs for each step:

have ‹ *in-dropzone* ∧ ¬ (*B at-block*) ∧ ¬ (*B holding*) ∧ *G collect* ↦ *B in-room* ∧ ¬ (*B at-block*) ∧ ¬ (*B holding*) ∧ *G collect*›
...
moreover have ‹*B in-room* ∧ ¬ (*B at-block*) ∧ ¬ (*B holding*) ∧ *G collect* ↦ *B in-room* ∧ *B at-block* ∧ ¬ (*B holding*) ∧ *G collect*›
...
moreover have ‹*B in-room* ∧ *B at-block* ∧ ¬ (*B holding*) ∧ *G collect* ↦ *B in-room* ∧ *B holding* ∧ *G collect*›
...
moreover have ‹*B in-room* ∧ *B holding* ∧ *G collect* ↦ *B in-dropzone* ∧ *B holding* ∧ *G collect*›
...
moreover have ‹*B in-dropzone* ∧ *B holding* ∧ *G collect* ↦ *B collect*›
...
ultimately show *?thesis* **using** *imp* **by** *blast*

Every step in the proof above is achieved by a single action; again this is due to the simplicity of the example. We can easily conceive of programs which have multiple paths to the desired state in which case all paths need to be considered.

Due to Lemma 5.6.3, the proof of *B in-dropzone* ∧ ¬ (*B at-block*) ∧ ¬ (*B holding*) ∧ *G collect* ↦ *B collect* shows a correctness property of our simple example agent.

5.8 Conclusions

We have presented a formalization of a verification framework for GOAL agents in its entirety. Furthermore, we have demonstrated how it can be applied to an agent specified in the agent logic by means of a simple example.

There are still a number of issues for our attention in the future. As was clear in our earlier work, the original formulation of the verification framework has some limitations. First, the framework is limited to single agent programs. Long term, the aim is to verify programs with multiple communicating agents. To this end, we want to explore how existing work on extending the framework can be integrated into our Isabelle/HOL formalization [16]. Second, the agent logic is limited to propositional logic which not only causes an inconvenience in specifying complex agent programs, but also means that certain things cannot be modeled.

Outside of extending the framework, there are also a number of usability concerns to address. This has little theoretical significance, but it would be interesting to experiment with providing more means of automation for conducting proofs of agent properties. While we did not have the space to show the proofs of our example agent in full details, we note that it can become quite tedious to conduct these proofs manually. Due to the complexity of the structures involved in the proofs, the automation of Isabelle/HOL is not geared towards such proofs out-of-the-box. Further work is required to enable more automation.

Nevertheless, the present paper demonstrates that a theorem proving approach to verifying agents is feasible. Furthermore, because we have formalized the framework in a proof assistant, we are able to provide a high level of assurance as everything is checked by Isabelle/HOL. In conclusion, we are excited to see the future potential of the framework enabled by the capabilities of Isabelle/HOL.

Acknowledgements

Koen V. Hindriks has provided valuable insights into the details of GOAL. Jørgen Villadsen, Frederik Krogsdal Jacobsen and Asta Halkjær From have commented on drafts.

Chapter 6

GOAL in Isabelle/HOL: Selected Proofs

In this chapter, we go into details with two selected proofs from the Isabelle/HOL formalization of the verification framework presented in Chapter 5. Neither of our papers [41, 44, 46, 47] devote their attention to describing the intricate details of proofs. The reason is that the papers have been limited by the number of available pages, and the proofs require extensive space to present and describe. Their focus have thus been on the overall picture and the most important results.

Note that the syntax has been simplified for readability. The source files may be consulted for the full details [45].

6.1 Proof of Selected Theorem

Theorem 5.6.1 corresponds to Theorem 4.15 in the seminal paper by de Boer, Hindriks, van der Hoek and Meyer [22]. The structure of the "pen and paper" proof translates well to Isar (the Isabelle/HOL language for structured proofs).

The theorem is integral to the verification method as it states that safety properties concerning the *unless* operator are equivalent to a finite set of Hoare triples. See Section 5.4.3 for the semantics of Hoare triples and Section 5.6 for the semantics of temporal logic.

theorem
$\forall (v \triangleright do\ b) \in \Pi.\ \models_H \{\ \varphi \wedge \neg\ \psi\ \}\ (v \triangleright do\ b)\ \{\ \varphi \vee \psi\ \})$
$\longleftrightarrow$ *Agent* $\models_T \varphi$ *unless* ψ

We apply the default proof method which produces subgoals for proving each direction separately.

proof
assume $*$: $\forall (v \triangleright do\ b) \in \Pi.\ (\models_H \{\ \varphi \wedge \neg\ \psi\ \}\ (v \triangleright do\ b)\ \{\ \varphi \vee \psi\ \})$
show *Agent* $\models_T \varphi$ *unless* ψ
...
next
assume *Agent* $\models_T \varphi$ *unless* ψ
show $\forall\ b \in \Pi.\ (\models_H \{\ \varphi \wedge \neg\ \psi\ \}\ (v \triangleright do\ b)\ \{\ \varphi \vee \psi\ \})$
...
qed

We start by proving the right direction. The proof is by contradiction, so we start by assuming that the *unless* property does not hold, and the goal is to show *False*.

assume $*$: $\forall (v \triangleright do\ b) \in \Pi.\ (\models_H \{\ \varphi \wedge \neg\ \psi\ \}\ (v \triangleright do\ b)\ \{\ \varphi \vee \psi\ \})$
show *Agent* $\models_T \varphi$ *unless* ψ
proof (*rule ccontr*)
assume $\neg$ *Agent* $\models_T \varphi$ *unless* ψ
...
then show *False*
...
qed

The *Agent* is defined as the set of all fair traces from the initial state, so from the assumption, we realize that for some trace $s \in$ *Agent*, at some step i the property does not hold.

assume $\neg$ *Agent* $\models_T \varphi$ *unless* ψ
then obtain $s\ i$ **where** $\neg\ (s,\ i \models_T \varphi$ *unless* $\psi)\ s \in$ *Agent* **by** *auto*

We then unfold the definition of *unless*, and due to the semantics we can conclude that either (1) ψ never becomes true and at some φ becomes false, or (2) at some point φ becomes false and later ψ becomes true.

then have $s,\ i \models_T \varphi \neg\ (s,\ i \models_T \varphi\ \mathit{until}\ \psi)$ **unfolding** *unless-def* **by** *simp-all*
then have
$\neg(\exists\ j \geq i.\ s,\ j \models_T \psi) \wedge (\exists\, k \geq i.\ s,\ k \models_T (\neg_T\ \varphi)) \vee$
$(\exists\ j > i.\ s,\ j \models_T \psi) \wedge (\forall\ j > i.\ s,\ j \models_T \psi \longrightarrow \neg(\forall\, k \geq i.\ j > k \longrightarrow s,\ k \models_T \varphi))$
by *auto*

We then show that each disjunct leads to *False*.

then show *False*
proof
assume *a*: $\neg\ (\exists\, j{\geq}i.\ s,\ j \models_T \psi) \wedge (\exists\, k{\geq}i.\ s,\ k \models_T (\neg_T\ \varphi))$
...
show *False* ...
next
assume *a*:
$(\exists\ j > i.\ s,\ j \models_T \psi) \wedge (\forall\ j > i.\ s,\ j \models_T \psi \longrightarrow \neg(\forall\, k \geq i.\ j > k \longrightarrow s,\ k \models_T \varphi))$
...
show *False*
...
qed

We define a lambda function *?P* which checks whether φ is false at a step $x \geq i$.

assume *a*: $\neg\ (\exists\, j{\geq}i.\ s,\ j \models_T \psi) \wedge (\exists\, k{\geq}i.\ s,\ k \models_T (\neg_T\ \varphi))$
let *?P* $= \lambda x.\ x \geq i \wedge s,\ x \models_T (\neg_T\ \varphi)$

φ can never be false at step *0*, but from our assumption, it must be the case that it eventually becomes false at some step *n*.

have $\neg$ *?P 0* **using** $s,\ i \models_T \varphi$ **by** *fastforce*
moreover obtain *n* **where** *?P n* **using** *a* **by** *auto*

From these two facts, we can show that φ will remain true from step i until some step $k > i$.

ultimately have $\exists\, k{\leq}n.\ (\forall\, i{<}k.\ \neg\ ?P\ i) \wedge\ ?P\ k$
using *ex-least-nat-le*[**where** *P*=*?P*] **by** *simp*
then obtain *k* **where** *Pk*: $k \leq n \wedge (\forall\, i{<}k.\ \neg\ ?P\ i) \wedge\ ?P\ k$ **by** *auto*
moreover from *this* **have** $k > i$ **using** $s,\ i \models_T \varphi$ **by** *fastforce*

We then conclude that φ is true at step $k - 1$.

ultimately have $\neg$ *?P* (*k* − *1*) **by** *simp*

From the assumption *a*, we further conclude that ψ is false at step *k* −*1*.

with $k > i$ **have** $s, k - 1 \models_T (\varphi \wedge_T \neg_T \psi)$ **using** *a* **by** *simp*
then have *st-nth s* (*k* − *1*) $\models_M \varphi \wedge \neg \psi$
by (*simp add*: $sem_{T\,M}$*-equiv2 transfer-semantics*$_M$)

In the following step *k*, φ has become false, but ψ remains false.

moreover have *?P k* **using** *Pk* **by** *simp*
then have $s, k \models_T (\neg_T \varphi \wedge_T \neg_T \psi)$ **using** *a* $k > i$ **by** *simp*
then have *st-nth s k* $\models_M \neg \varphi \wedge \neg \psi$
by (*simp add*: $sem_{T\,M}$*-equiv2 transfer-semantics*$_M$)

What remains is to show that this contradicts our initial assumption $*$.

moreover obtain v *b* **where** $(v \triangleright do\ b) = act\text{-}nth\ s\ (k - 1)$ **by** *simp*
moreover from *this* **have** $(v \triangleright do\ b) \in \Pi$
using $s \in Agent$ *trace-in-*Π **unfolding** *Agent-def* **by** *simp*
with $*$ **have** $\forall\, s \in Agent.\ \forall\, i.\ ((\varphi \wedge \neg \psi)[s\ i]_M) \wedge (v \triangleright do\ b) = (act\text{-}nth\ s\ i) \longrightarrow$
$((\varphi \vee \psi)[s\ (i+1)]_M)$
using $semantics_H$*.simps*(*2*) **by** *blast*
then have $((\varphi \wedge \neg \psi)[s\ (k-1)]_M) \wedge (v \triangleright do\ b) = (act\text{-}nth\ s\ (k-1)) \longrightarrow$
$((\varphi \vee \psi)[s\ ((k-1)+1)]_M)$
using $s \in Agent$ **by** *simp*
with $k > i$ **have** $((\varphi \wedge \neg \psi)[s\ (k-1)]_M) \wedge (v \triangleright do\ b) = (act\text{-}nth\ s\ (k-1)) \longrightarrow$
$((\varphi \vee \psi)[s\ k]_M)$
by *simp*
ultimately show *False* **by** *simp*

We now need to show that also the other disjunct leads to *False*.

assume *a*:
$(\exists\ j > i.\ s, j \models_T \psi) \wedge (\forall\ j > i.\ s, j \models_T \psi \longrightarrow \neg(\forall\, k \geq i.\ j > k \longrightarrow s, k \models_T \varphi))$

We define a lambda function *?P* which checks whether ψ is true at a step $x > i$.

let *?P* = $\lambda x.\ x > i \wedge s, x \models_T \psi$

Again, we immediately note that ψ does not hold at a step *0* (greater than i), but that it will eventually hold at some step n.

have $\neg$ *?P 0* **by** *simp*
moreover obtain n **where** *?P n* **using** *a* **by** *auto*

From these two facts, it must be the case that ψ is false at all steps between i and some step $k \leq n$.

ultimately have $\exists k{\leq}n.\ (\forall i{<}k.\ \neg$ *?P i*$) \wedge$ *?P k*
  using *ex-least-nat-le*[**where** *P=?P*] **by** *simp*
then obtain k **where** *Pk*: $k \leq n \wedge (\forall i{<}k.\ \neg$ *?P i*$) \wedge$ *?P k* **by** *auto*

We need further help from a lambda function *?P′* which checks whether φ is true at a step $i < x <k$.

let *?P′* $= \lambda x.\ i < x \wedge x < k \wedge \neg\ s,\ x \models_T \varphi$

From the assumption *a* and the fact *?P k*, we conclude that *?P* holds for step *0* and some step n'.

have $\neg$ *?P′ 0* **by** *simp*
moreover obtain n' **where** *?P′ n′* **using** *Pk s*, $i \models_T \varphi$ *le-neq-trans a* **by** *blast*

From these two facts, it must be the case that φ becomes false at some step j where $i < j <k$.

ultimately have $\exists j{\leq}n'.\ (\forall i{<}j.\ \neg$ *?P′ i*$) \wedge$ *?P′ j*
  using *ex-least-nat-le*[**where** *P=?P′*] **by** *simp*
then obtain j **where** *Pj*: $j \leq n' \wedge (\forall i{<}j.\ \neg$ *?P′ i*$) \wedge$ *?P′ j* **by** *auto*
then have $\neg$ *?P′* $(j - 1)$ **by** *simp*
moreover have $j - 1 < k$ **using** *Pj* **by** *auto*
ultimately have $s,\ j - 1 \models_T \varphi$
proof (*cases* $i < j - 1$)
  case *False*
  then have $i = j - 1$ **using** *Pj* **by** *auto*
  with $s,\ i \models_T \varphi$ **show** *?thesis* **by** *simp*
qed *simp*

It must be the case that both ψ and φ are false at step $j - 1$.

moreover have $\neg$ *?P* $(j - 1)$ **using** *Pk* **and** $j - 1 < k$ **by** *blast*
then have $s, j - 1 \models_T (\neg_T \psi)$
proof (*cases* $i < j - 1$)
  case *False*
  then have $i = j - 1$ **using** *Pj* **by** *auto*
  with $\neg$ s, $i \models_T \varphi$ *until* ψ **show** *?thesis* **by** *fastforce*
qed *simp*

Again, what remains is to show that this contradicts our initial assumption $*$.

ultimately have $s, j - 1 \models_T (\varphi \wedge_T \neg_T \psi)$ **by** *simp*
then have *st-nth* s $(j - 1) \models_M \varphi \wedge \neg \psi$
  by (*simp add*: $sem_{T\,M}$*-equiv2 transfer-semantics*$_M$)
moreover
{
  have $\neg$ $s, j \models_T \varphi$ **using** *Pj* **by** *simp*
  moreover have $j < k$ **using** *Pj* **by** *simp*
  then have $\neg$ *?P j* **using** *Pk* **by** *simp*
  then have $\neg$ $s, j \models_T \psi$ **using** *Pj* **by** *simp*
  ultimately have $s, j \models_T (\neg_T \varphi \wedge_T \neg_T \psi)$ **by** *simp*
  then have *st-nth s j* $\models_M \neg \varphi \wedge \neg \psi$
    by (*simp add*: $sem_{T\,M}$*-equiv2 transfer-semantics*$_M$)
}
moreover obtain υ b **where** $(\upsilon \triangleright do\ b) = act\text{-}nth\ s\ (j - 1)$ **by** *simp*
moreover from *this* **have** $(\upsilon \triangleright do\ b) \in \Pi$
  using $s \in Agent$ *trace-in-*Π **unfolding** *Agent-def* **by** *simp*
with $*$ **have** $\forall\ s \in Agent.\ \forall i.\ ((\varphi \wedge \neg \psi)[s\ i]_M) \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ i) \longrightarrow$
  $((\varphi \vee \psi)[s\ (i+1)]_M)$
using $semantics_H.simps(2)$ **by** *blast*
then have $((\varphi \wedge \neg \psi)[s\ (j-1)]_M) \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ (j-1)) \longrightarrow$
  $((\varphi \vee \psi)[s\ ((j-1)+1)]_M)$
using $s \in Agent$ **by** *simp*
with *Pj* **have** $((\varphi \wedge \neg \psi)[s\ (j-1)]_M) \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ (j-1)) \longrightarrow$
  $((\varphi \vee \psi)[s\ j]_M)$
by *simp*
ultimately show *False* **by** *simp*

This concludes our proof of the right direction, and we turn our attention to the left direction, where we assume the temporal property to hold and show that it leads to the truth of a finite set of Hoare triples.

The default proof strategy for a universal quantifier is to fix the universally quantified variable such that we can work with the formula without the quantifier. The proof goal has three nested quantifiers (two of them are hidden).

assume *Agent* $\models_T \varphi$ *unless* ψ
show $\forall b \in \Pi.\ (\models_H \{\ \varphi \wedge \neg\ \psi\ \}\ (\upsilon \triangleright do\ b)\ \{\ \varphi \vee \psi\ \})$
proof
  fix b
  assume $b \in \Pi$
  show $\models_H \{\ \varphi \wedge \neg\ \psi\ \}\ (\upsilon \triangleright do\ b)\ \{\ \varphi \vee \psi\ \}$
  proof −
    have $\forall s \in Agent.\ \forall i.\ ((\varphi \wedge \neg\ \psi)[s\ i]_M) \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ i) \longrightarrow$
      $((\varphi \vee \psi)[s\ (i{+}1)]_M)$
    proof
      fix s
      assume $s \in Agent$
      show $\forall i.\ ((\varphi \wedge \neg\ \psi)[s\ i]_M) \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ i) \longrightarrow$
        $((\varphi \vee \psi)[s\ (i{+}1)]_M)$
      proof
        fix i
        show $((\varphi \wedge \neg\ \psi)[s\ i]_M) \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ i) \longrightarrow$
          $((\varphi \vee \psi)[s\ (i{+}1)]_M)$
        ...
      qed
    qed
    then show *?thesis* **using** $semantics_H.simps(2)$ **by** *blast*
  qed
qed

What remains is to prove the truth of a Hoare triple (containing the fixed variables). The proof is by case distinction on whether φ is true in step i.

show $((\varphi \wedge \neg\ \psi)[s\ i]_M) \wedge (\upsilon \triangleright do\ b) = (act\text{-}nth\ s\ i) \longrightarrow ((\varphi \vee \psi)[s\ (i{+}1)]_M)$
proof (*cases st-nth s i* $\models_M \varphi$)
  case *True*
  ...
  show *?thesis*
  ...
next
  case *False*
  ...
  show *?thesis*
  ...
qed

The case that φ is false is solved immediately by automation, and we focus on the case that it is true. First, we conclude that φ is true at step i in trace s when formulated in temporal logic.

case *True*
then have $s,\ i \models_T \varphi$ **by** (*simp add*: sem_{TM}*-equiv2 transfer-semantics*$_M$)

From our initial assumption, we conclude that φ *until* ψ must be true at step i.

with *Agent* $\models_T \varphi$ *unless* ψ **have** $s,\ i \models_T \varphi$ *until* ψ
  using $s \in$ *Agent* **unfolding** *unless-def* **by** *simp*

Using the semantics of *until*, it must then be the case that φ remains true at least until ψ becomes true at some step j, or φ remains true forever.

then have $(\exists\ j \geq i.\ (s,\ j \models_T \psi) \wedge (\forall k \geq i.\ j > k \longrightarrow (s,\ k \models_T \varphi))) \vee$
  $(\forall k \geq i.\ (s,\ k \models_T \varphi))$ **by** *simp*
then obtain j **where** $*$: $(j \geq i \wedge s,\ j \models_T \psi \wedge (\forall k \geq i.\ j > k \longrightarrow s,\ k \models_T \varphi)) \vee$
$s,\ i + 1 \models_T \varphi$ **by** *auto*

We then distinguish between whether $j < i + 1$. We first consider the case that it is true.

then show *?thesis*
proof (*cases* $j > i + 1$)
  case *True*
  ...
  show *?thesis*
  ...
next
  case *False*
  ...
  show *?thesis*
  ...
qed

We first need to consider the case that $j < i + 1$. Using the semantics of *until*, we conclude that φ is true in step $i + 1$, and the postcondition is true.

case *True*
with $*$ **have** $s,\ (i + 1) \models_T \varphi$ **by** *auto*
then have *st-nth s* $(i + 1) \models_M \varphi \vee \psi$
  by (*simp add*: sem_{TM}*-equiv2 transfer-semantics*$_M$)
then show *?thesis* **by** *simp*

We then need to consider the case that $j \geq i + 1$. Using the semantics of *until*, it must then be the case that either (1) ψ is true at step i, (2) ψ is true at step $i + 1$, or (3) φ is true at step $i + 1$.

case *False*
with $*$ **have** $**$: s, $i \models_T \psi \vee s$, $i + 1 \models_T \psi \vee s$, $i + 1 \models_T \varphi$
  by (*metis discrete order.not-eq-order-implies-strict*)

If ψ is true at step i, the precondition of the Hoare triple is not satisfied, and it is thus true. Otherwise, we know that either φ or ψ is true in the following step, and the postcondition is true. $i + 1$, or (3) φ is true at step $i + 1$.

then show *?thesis*
proof (*cases st-nth s i* $\models_M \psi$)
  case *False*
  then have $\neg$ s, $i \models_T \psi$ **by** (*simp add*: $sem_{T\,M}$*-equiv2 transfer-semantics*$_M$)
  with $**$ **have** s, $i + 1 \models_T \psi \vee s$, $i + 1 \models_T \varphi$ **by** *simp*
  then show *?thesis* **using** $sem_{T\,M}$*-equiv2 transfer-semantics*$_M$ **by** *auto*
qed *simp*

This concludes the proof in the left direction, which means we have now shown both directions, and we can conclude that the *unless* operator is equivalent to a finite set of Hoare triples.

6.2 Proof of Selected Lemma

Lemma 5.4.4 corresponds to Lemma 4.3 in the seminal paper by de Boer, Hindriks, van der Hoek and Meyer [22]. In the formal semantics of GOAL, there is a clear relation between executing basic actions and conditional actions. Lemma 5.4.4 makes it precise how the two types of actions are related in the Hoare logic.

lemma
  $\models_H$ **{** $\varphi \wedge \psi$ **}** a **{** φ' **}** $\Longrightarrow \forall\ s \in Agent.\ \forall\, i.\ ((\varphi \wedge \neg\psi) \longrightarrow \varphi')[s\ i] \Longrightarrow$
    $\models_H$ **{** φ **}** $(\psi \triangleright do\ a)$ **{** φ' **}**

We first unfold the semantics of Hoare triples for conditional actions.

proof −
  have $\forall\ s \in \mathit{Agent}.\ \forall i.\ (\varphi[s\ i]_M \wedge (\psi \triangleright \mathit{do}\ a) = (\mathit{act\text{-}nth}\ s\ i) \longrightarrow (\varphi'[s\ (i{+}1)]_M))$
  ...
  then show *?thesis* **by** *simp*
qed

Note that $\varphi[s\ i]_M$ evaluates the mental state formula φ in step i of trace s. The equality $(\psi \triangleright \mathit{do}\ a) = (\mathit{act\text{-}nth}\ s\ i)$ ensures that the stated conditional action is in fact the selected action on this step in the trace.

We fix a trace s and by definition it is a trace of the agent (*is-trace s*). This allows us to focus on the truth of the Hoare triple at a fixed step i in the trace.

fix *s*
assume $s \in \mathit{Agent}$
then have *is-trace s* **unfolding** *Agent-def* **by** *simp*
show $\forall i.\ (\varphi[s\ i]_M \wedge (\psi \triangleright \mathit{do}\ a) = \mathit{act\text{-}nth}\ s\ i \longrightarrow (\varphi'[s\ (i{+}1)]_M))$
proof
  fix *i*
  show $\varphi[s\ i]_M \wedge (\psi \triangleright \mathit{do}\ a) = \mathit{act\text{-}nth}\ s\ i \longrightarrow (\varphi'[s\ (i{+}1)]_M)$
  ...
qed

We assume the left-hand side of the implication, and then prove the right-hand side for both cases of $\psi[s\ i]_M$.

assume $\varphi[s\ i]_M \wedge (\psi \triangleright \mathit{do}\ a) = \mathit{act\text{-}nth}\ s\ i$
with *conjunct2* **have** $(\psi \triangleright \mathit{do}\ a) = \mathit{act\text{-}nth}\ s\ i$ **.**
show $\varphi'[s\ (i{+}1)]_M$
proof (*cases* $\psi[s\ i]_M$)
  case *False*
  ...
  show *?thesis*
  ...
next
  case *True*
  ...
  show *?thesis*
  ...
qed

We first show the case that $\psi[s\ i]$ is false. From this, it follows that we have *st-nth s (i+1) = st-nth s i*. We have previously assumed $((\varphi \wedge \neg\psi) \longrightarrow \varphi')[s\ i]$ and $(\varphi \wedge \neg\psi)\ [s\ i]$, so we can conclude $\varphi'[s\ i]$.

case *c*: *False*
then have ¬ ((*st-nth s i*) →(*ψ* ▷ *do a*) (*st-nth s* (*i*+*1*)))
  unfolding *transition-def* **by** *simp*
with *is-trace s* (*ψ* ▷ *do a*) = *act-nth s i* **have** (*st-nth s* (*i*+*1*)) = (*st-nth s i*)
  using *not-transition-eq* **by** *metis*
moreover from *assms*(*2*) **have** (*st-nth s i*) ⊨$_M$ (*φ* ∧ ¬*ψ*) ⟶ *φ*′
  using *s* ∈ *Agent* **by** *simp*
with *c φ*[*s i*]$_M$ ∧ (*ψ* ▷ *do a*) = *act-nth s i* **have** (*st-nth s i*) ⊨$_M$ *φ*′ **by** *simp*
ultimately show *?thesis* **by** *simp*

We then consider the case that *ψ*[*s i*] is true. The semantics of Hoare triples for conditional actions are specified using traces, while the semantics of Hoare triples for basic actions are specified in the mental state transformer semantics. The tricky part is that starting from our assumption about the semantics of Hoare triples for basic actions, we need to show the semantics of Hoare triples for conditional actions.

case *c*: *True*
from *φ*[*s i*]$_M$ ∧ (*ψ* ▷ *do c*) = *act-nth s i* **have** (*st-nth s i*) ⊨$_M$ *φ* **by** *simp*
moreover from *a* **have** (*st-nth s i*) ⊨$_M$ *ψ* **by** *simp*
ultimately have (*st-nth s i*) ⊨$_E$ (*φ* ∧ *ψ*) **using** *transfer-semantics*$_M$ **by** *simp*
moreover from *mst-reachable-basic-trace* **have** *mst-reachable-basic* (*st-nth s i*)
  using *s* ∈ *Agent* **by** *auto*
then have
  ((*st-nth s i*) ⊨$_E$ (*φ* ∧ *ψ*) ∧ (*enabledb a*) ⟶ *the* (*M a* (*st-nth s i*)) ⊨$_M$ *φ*′) ∧
    ((*st-nth s i*) ⊨$_E$ (*φ* ∧ *ψ*) ∧ ¬(*enabledb a*) ⟶ (*st-nth s i*) ⊨$_M$ *φ*′)
  using *assms*(*1*) *semantics*$_H$.*simps*(*1*) **by** *blast*
ultimately have ∗:
  ((*st-nth s i*) ⊨$_E$ (*enabledb a*) ⟶ *the* (*M a* (*st-nth s i*)) ⊨$_M$ *φ*′) ∧
    ((*st-nth s i*) ⊨$_E$ ¬(*enabledb a*) ⟶ (*st-nth s i*) ⊨$_M$ *φ*′) **by** *simp*

We have now established that the enabledness of the action at step *i* determines whether the postcondition should be evaluated in the same state or in the state that follows from executing the action. We consider the two cases separately, starting with the case that the action is enabled.

show *?thesis*
proof (*cases* (*st-nth s i*) ⊨$_E$ *enabledb a*)
  case *enabled*: *True*

When the action is enabled, we know that the postcondition is true in the state that follows from executing the action. What remains is to convince Isabelle

that the result of applying the mental state transformer is in fact the next step in the trace.

with $*$ **have** *the* $(\mathcal{M}\ a\ (st\text{-}nth\ s\ i)) \models_M \varphi'$ **by** *simp*
moreover from *enabled* **have** $\mathcal{M}\ a\ (st\text{-}nth\ s\ i) \neq None$ **by** $(cases\ a)$ *simp-all*
ultimately show $(st\text{-}nth\ s\ (i{+}1)) \models_M \varphi'$
  using $\mathcal{M}\text{-}suc\text{-}state\ s \in Agent\ (\psi \triangleright do\ a) = act\text{-}nth\ s\ i$
using *snd-act-nth* **using** *a fst-act-nth* **by** *auto*

We still need to consider the case that the action is not enabled. When the action is not enabled, we know that the postcondition is true in the same state. Due to the difference between the semantics of the two types of Hoare triples, we need to show that no mental state follows from applying the mental state transformer. We consider each of the three types of basic actions individually.

case *not-enabled*: *False*
with $*$ **have** $(st\text{-}nth\ s\ i) \models_M \varphi'$ **by** *simp*
moreover have $\mathcal{M}\ a\ (st\text{-}nth\ s\ i) = None$
proof $(cases\ a)$

For all cases, $\mathcal{M}\ a\ (st\text{-}nth\ s\ i) = None$ follows directly from the semantics once we unfold the definitions. We show here only the first case.

case $(basic\ n)$
with *not-enabled* **have** $\neg\ ((enabledb\ (basic\ n))[s\ i]_E)$ **by** *simp*
with $semantics_E'.simps(3)$ **have** $\neg\ \mathcal{M}\ (basic\ n)\ (st\text{-}nth\ s\ i) \neq None$
  using $\mathcal{M}\text{-}some$ **by** *fastforce*
then have $\mathcal{M}\ (basic\ n)\ (st\text{-}nth\ s\ i) = None$ **by** *blast*
with *basic* **show** *?thesis* **by** *simp*

Showing $\varphi'[s\ (i{+}1)]_M$ is now simple, as the definition of traces allow us to conclude $st\text{-}nth\ s\ (i{+}1) = st\text{-}nth\ s\ i$.

then have $\neg((st\text{-}nth\ s\ i) \rightarrow(\psi \triangleright do\ a)\ (st\text{-}nth\ s\ (i{+}1)))$
  unfolding *transition-def* **by** *simp*
with $is\text{-}trace\ s\ (\psi \triangleright do\ a) = act\text{-}nth\ s\ i$ **have** $(st\text{-}nth\ s\ (i{+}1)) = (st\text{-}nth\ s\ i)$
  using *not-transition-eq* **by** *metis*
ultimately show *?thesis* **by** *simp*

This concludes our proof of the relation between Hoare triples for basic actions and conditional actions.

6.3 Concluding Remarks

In this section, we described in technical details the reasoning steps of two Isabelle proof developments. While the formalization contains various proofs, we selected Theorem 5.6.1 and Lemma 5.4.4 for two reasons. Firstly because the technical details provide a good insight into how proofs are developed in our Isabelle formalization. Secondly because they so closely follow their counterparts in the paper by de Boer, Hindriks, van der Hoek and Meyer [22], which provides a unique opportunity for a comparison between the "pen and paper" proofs and Isabelle proofs. While we do not go into details, our findings for the two selected proofs show that the Isabelle proof developments required substantially more work than the "pen and paper" versions suggested.

Appendix A

GOAL-DTU: Development of Distributed Intelligence for the Multi-Agent Programming Contest — Evaluation of Matches (Figures)

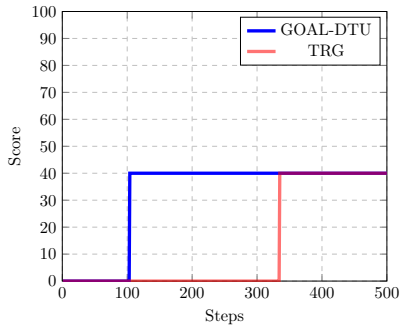

Figure A.1: Score vs. TRG (1)

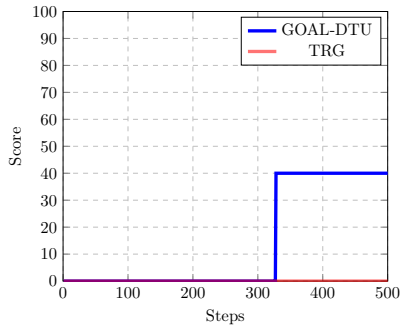

Figure A.2: Score vs. TRG (2)

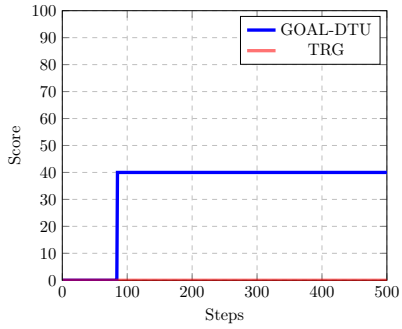

Figure A.3: Score vs. TRG (3)

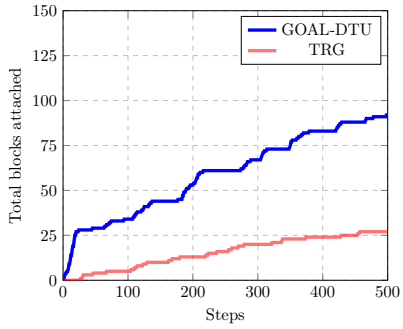

Figure A.4: Blocks vs. TRG (1)

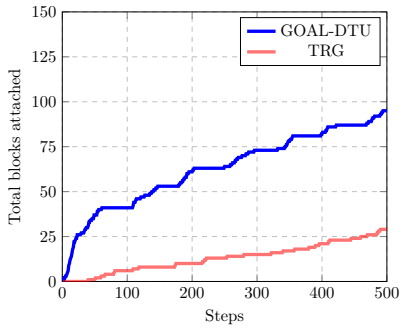

Figure A.5: Blocks vs. TRG (2)

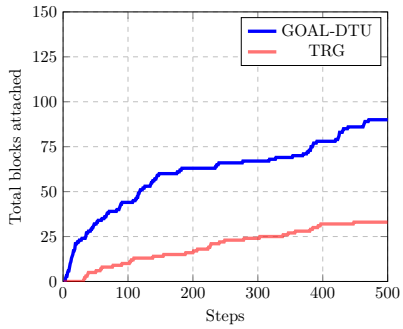

Figure A.6: Blocks vs. TRG (3)

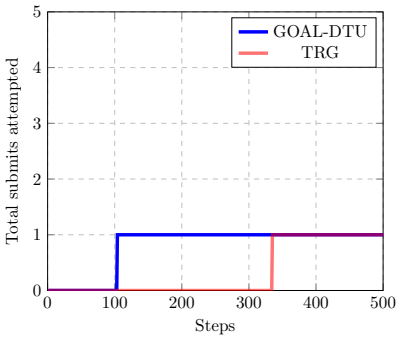

Figure A.7: Submit vs. TRG (1)

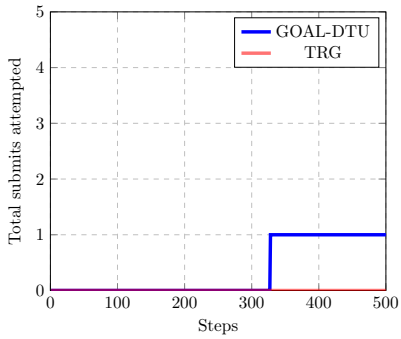

Figure A.8: Submit vs. TRG (2)

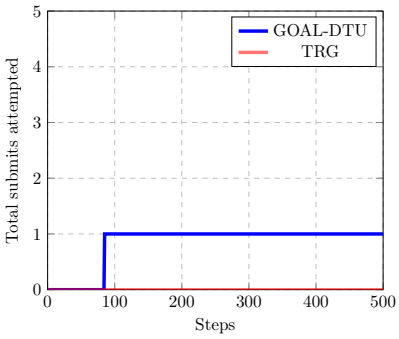

Figure A.9: Submit vs. TRG (3)

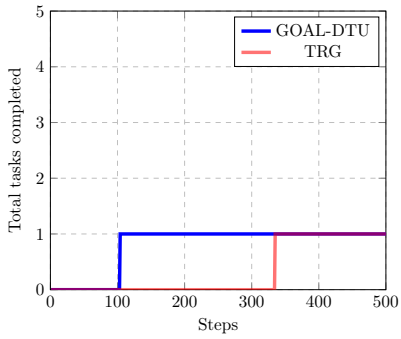

Figure A.10: Tasks vs. TRG (1)

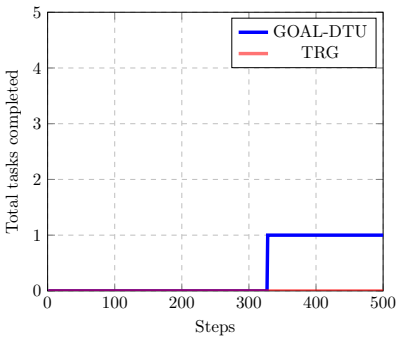

Figure A.11: Tasks vs. TRG (2)

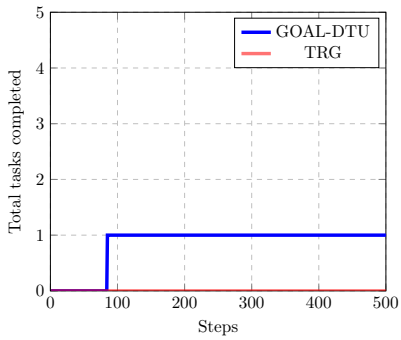

Figure A.12: Tasks vs. TRG (3)

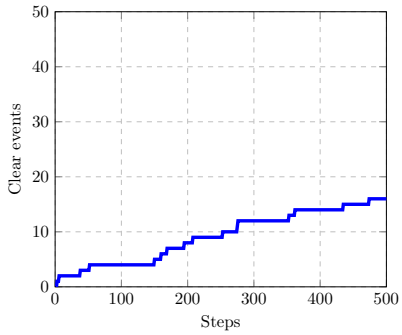

Figure A.13: Clear vs. TRG (1)

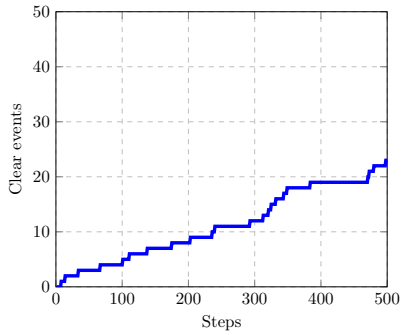

Figure A.14: Clear vs. TRG (2)

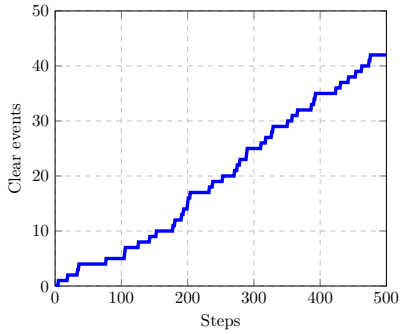

Figure A.15: Clear vs. TRG (3)

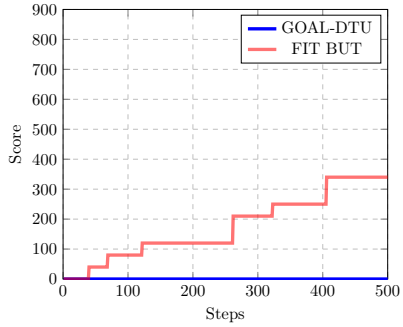

Figure A.16: Score vs. FIT BUT (1)

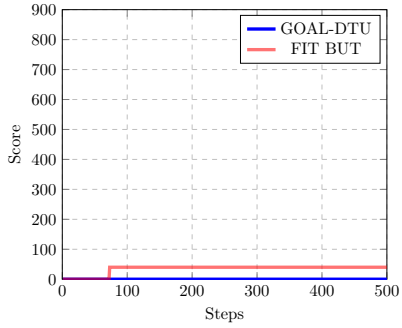

Figure A.17: Score vs. FIT BUT (2)

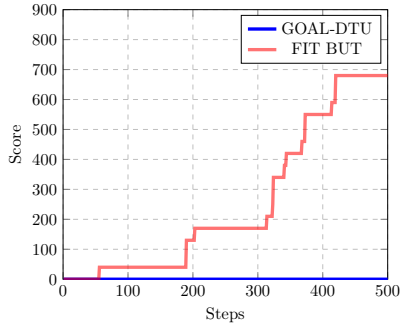

Figure A.18: Score vs. FIT BUT (3)

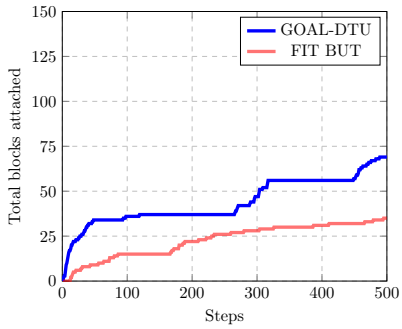

Figure A.19: Blocks vs. FIT BUT (1)

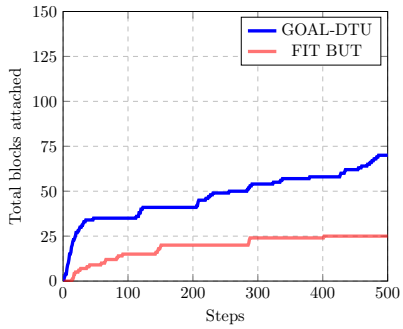

Figure A.20: Blocks vs. FIT BUT (2)

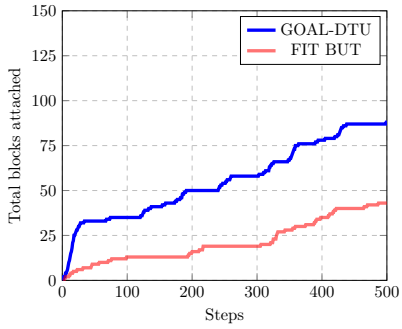

Figure A.21: Blocks vs. FIT BUT (3)

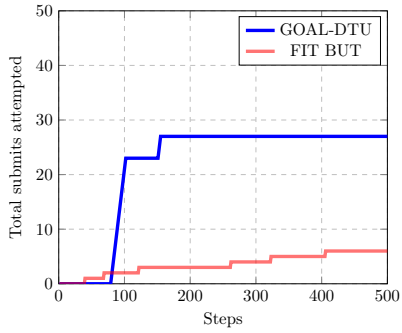

Figure A.22: Submit vs. FIT BUT (1)

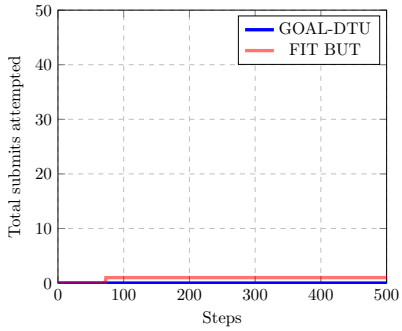

Figure A.23: Submit vs. FIT BUT (2)

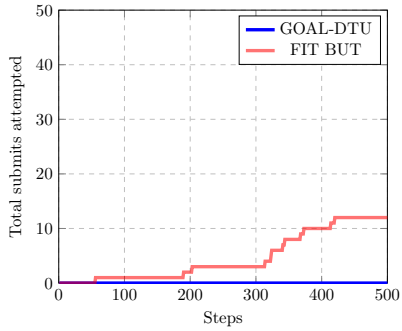

Figure A.24: Submit vs. FIT BUT (3)

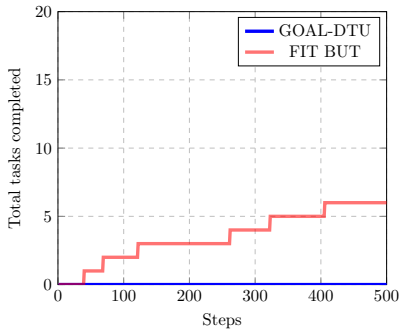

Figure A.25: Tasks vs. FIT BUT (1)

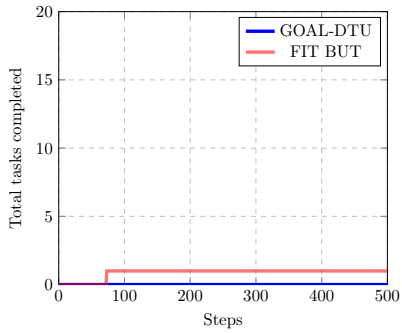

Figure A.26: Tasks vs. FIT BUT (2)

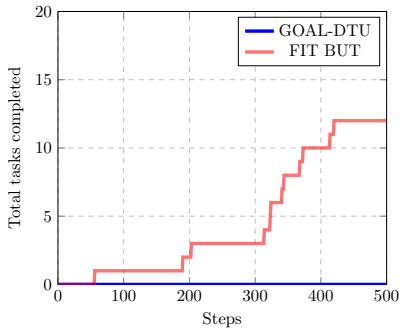

Figure A.27: Tasks vs. FIT BUT (3)

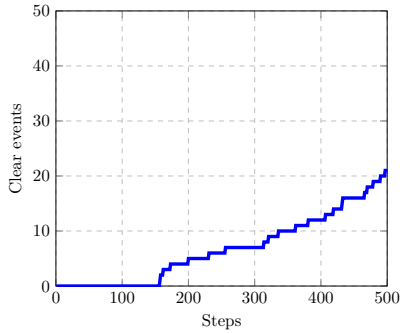

Figure A.28: Clear vs. FIT BUT (1)

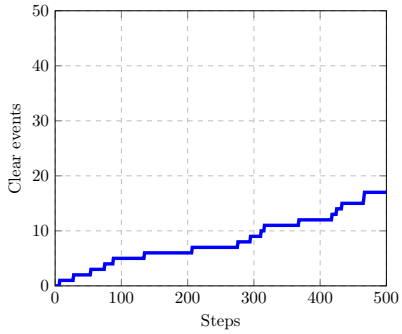

Figure A.29: Clear vs. FIT BUT (2)

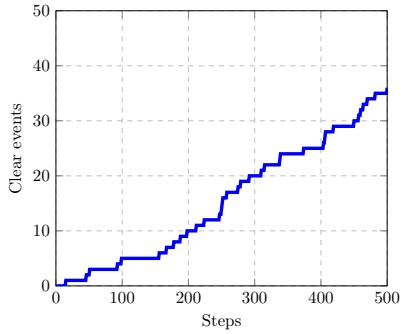

Figure A.30: Clear vs. FIT BUT (3)

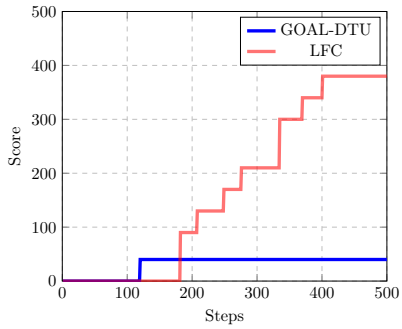

Figure A.31: Score vs. LFC (1)

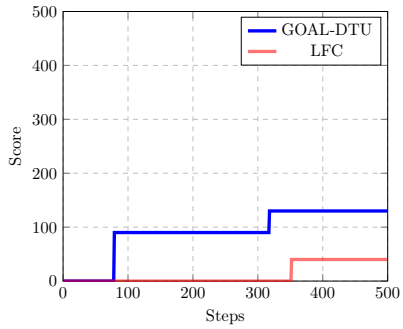

Figure A.32: Score vs. LFC (2)

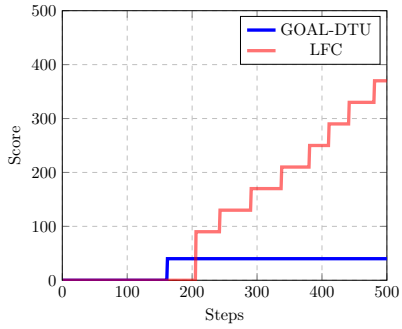

Figure A.33: Score vs. LFC (3)

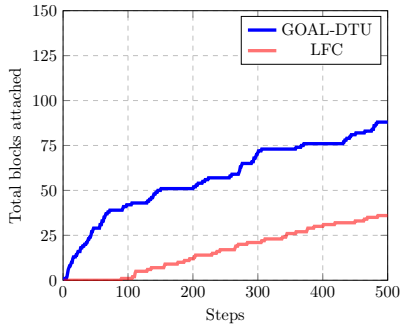

Figure A.34: Blocks vs. LFC (1)

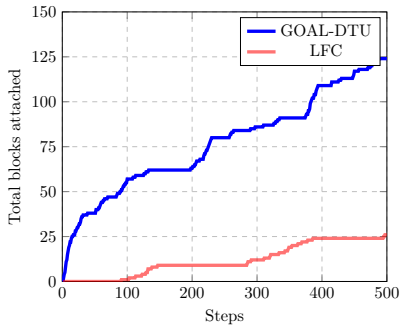

Figure A.35: Blocks vs. LFC (2)

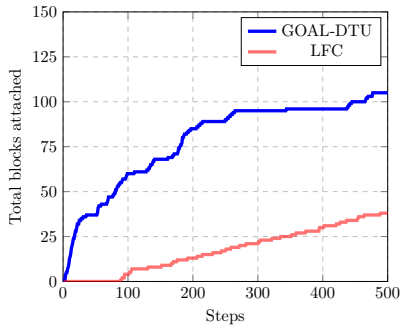

Figure A.36: Blocks vs. LFC (3)

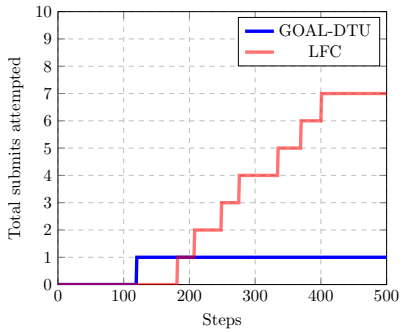

Figure A.37: Submit vs. LFC (1)

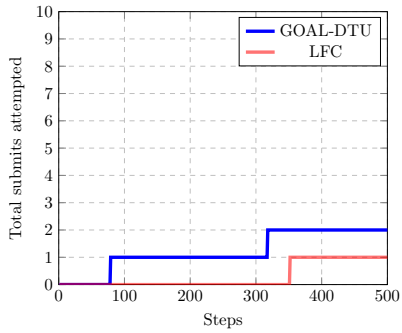

Figure A.38: Submit vs. LFC (2)

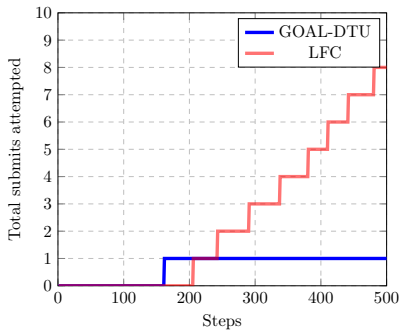

Figure A.39: Submit vs. LFC (3)

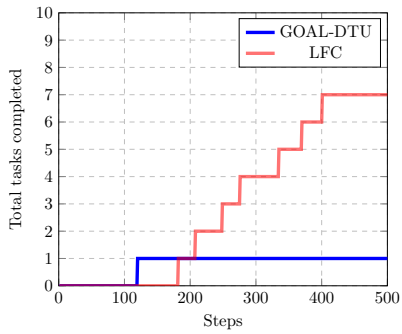

Figure A.40: Tasks vs. LFC (1)

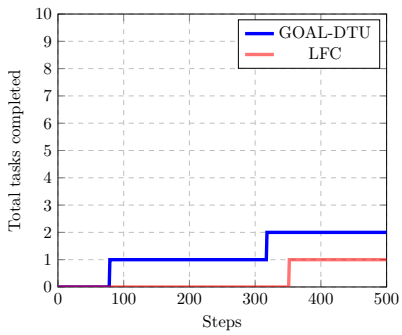

Figure A.41: Tasks vs. LFC (2)

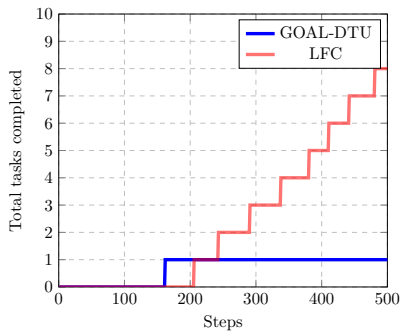

Figure A.42: Tasks vs. LFC (3)

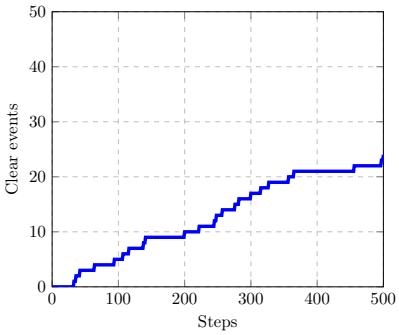

Figure A.43: Clear vs. LFC (1)

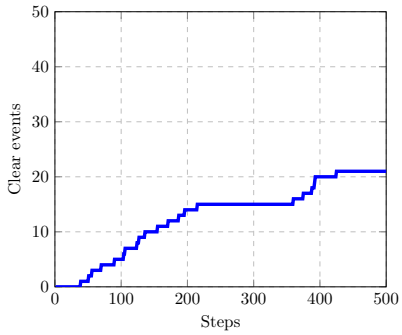

Figure A.44: Clear vs. LFC (2)

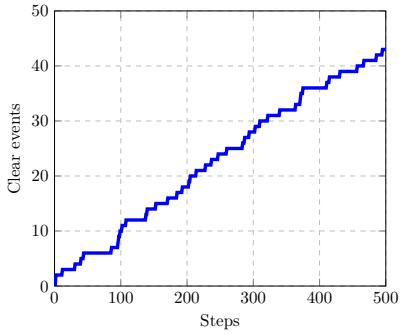

Figure A.45: Clear vs. LFC (3)

Appendix B

Towards Verifying a Blocks World for Teams GOAL Agent – Example

We consider here the first proof step: it reflects that the agent should first move from its initial position to the room containing the red block.

For the sake of readability, we introduce the shorthand φ_1 **ensures** ψ_1 for the first proof step (1):

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

Effect of $\boldsymbol{goTo}(r_1)$

We need to prove that $goTo(r_1)$ gives the desired state, i.e. the following Hoare triple must be satisfied:

$$\{\varphi_1 \wedge \neg\psi_1\}\ \mathit{enabled}(goTo(r_1)) \triangleright \mathit{do}(goTo(r_1))\ \{\psi_1\}$$

By applying the rule for conditional actions we are required to prove the formula:

$$(\varphi_1 \wedge \neg\psi_1 \wedge \neg\mathit{enabled}(goTo(r_1))) \longrightarrow \psi_1$$

We rewrite φ_1 and ψ_1 to their full definitions and immediately realize that the left-hand side is unsatisfiable. Additionally, we need to prove the Hoare triple:

$$\{\varphi_1 \wedge \neg\psi_1 \wedge \mathit{enabled}(\mathit{goTo}(r_1)\}\ \mathit{goTo}(r_1)\ \{\psi_1\}$$

For all actions a different from `goTo`, we supply the frame axiom

$$\{\mathsf{B}\mathit{in}(X)\}\ a\ \{\mathsf{B}\mathit{in}(X)\}$$

which captures that only the action `goTo` can change the agent's belief about its current position. Furthermore, for all actions a' different from `pickUp` or `putDown`, we supply the frame axiom

$$\{\neg\mathsf{B}\mathit{holding}(X)\}\ a'\ \{\neg\mathsf{B}\mathit{holding}(X)\}$$

which states that only the actions `pickUp` and `putDown` can change the agent's belief about blocks it is holding. Lastly, for all actions a'' different from `putDown`, we supply the frame axiom

$$\{\mathsf{G}\mathit{collect}(\mathit{red}) \wedge \mathsf{B}\mathit{color}(b_a, \mathit{red}, r_1)\}\ a'\ \{\mathsf{G}\mathit{collect}(\mathit{red}) \wedge \mathsf{B}\mathit{color}(b_a, \mathit{red}, r_1)\}$$

stating that the goal to collect a red block, and the information about the block, is unchanged by those actions. It may perhaps seem odd that picking up a block does not immediately change the belief about the position of the block, and this in a formalization detail we will not delve further into here.

We weaken the precondition and strengthen the postcondition by the consequence rule (and the invariant *inv-in*). We now need to prove the Hoare triple:

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

By the conjunction rule we split the proof into subproofs of the following three Hoare triples:

$$\{\mathsf{B}\mathit{in}(r_0) \wedge \neg\mathsf{B}\mathit{in}(r_1)\}\ \mathit{goTo}(r_1)\ \{\mathsf{B}\mathit{in}(r_1) \wedge \neg\mathsf{B}\mathit{in}(r_0)\}$$

$$\{\neg\mathsf{B}\mathit{holding}(b_a)\}\ \mathit{goTo}(r_1)\ \{\neg\mathsf{B}\mathit{holding}(b_a)\}$$

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

The first Hoare triple is the effect axiom for $\mathit{goTo}(r_1)$ that we derived in the transformation and the last two are frame axioms that we supplied above.

We need to show that other actions do not change the mental state since we only have a single trace. This plays well into our mutually exclusive decision rules. We will show it merely for a single action for completeness.

Non-effect of $goTo(r_0)$

To prove φ_1 **ensures** ψ_1, every action a should satisfy the Hoare triple $\{\varphi_1 \wedge \neg\psi_1\}\ a\ \{\varphi_1 \vee \psi_1\}$. For $goTo(r_0)$ we should thus prove:

$$\{\varphi_1 \wedge \neg\psi_1\}\ enabled(goTo(r_0)) \triangleright\ do(goTo(r_0))\ \{\varphi_1 \vee \psi_1\}$$

By the rule for conditional actions we need to assert the truth of the formula

$$(\varphi_1 \wedge \neg\psi_1 \wedge \neg enabled(goTo(r_0)) \longrightarrow (\varphi_1 \vee \psi_1)$$

which is easy to prove as the left-hand side of the implication is only true when φ_1 is true, which in turn guarantees the truth of the disjunction on the right-hand side. Furthermore, we must prove the Hoare triple:

$$\{\varphi_1 \wedge \neg\psi_1 \wedge enabled(goTo(r_0))\}\ goTo(r_0)\ \{\varphi_1 \vee \psi_1\}$$

By the consequence rule we weaken the precondition and strengthen the post-condition:

$$\begin{array}{l}\{\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge\ \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)\}\\ goTo(r_0)\\ \{\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge\ \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red)\}\end{array}$$

Essentially, the Hoare triple above states that the action has no effect on the mental state. The rule for infeasible actions allows us to prove this if we can instead prove the formula (using the $enabled(goTo(r_0)$ equivalence):

$$\begin{array}{l}\mathsf{B}color(b_a, red, r_1) \wedge \mathsf{B}in(r_0) \wedge \neg\mathsf{B}holding(b_a) \wedge \mathsf{G}collect(red) \longrightarrow\\ \neg(\mathsf{B}(holding(b_a) \wedge \neg in(r_0)))\end{array}$$

It is trivial to show that the formula is valid by considering the possible truth values of conjuncts on both sides of the implication.

Sketch of Remaining Proof Steps

We will merely sketch the remaining proof obligations: For the remainder of proof step (1), the proofs for other non-enabled actions are analogous as a result of mutually exclusive decision rules. For the proofs of steps (2)–(5), they are structurally similar to (1) but each for another enabled action and requiring additional invariants to be proved.

Appendix C

Changes to Published Articles

All published articles are included verbatim but have been recompiled in the present layout.

The following formatting changes apply to all published articles:

- The bibliographies have been merged.
- The abstract of each published article appears as the first paragraph of the corresponding chapter.
- Streamlined spelling of “model checking”, “straightforward”, “behavior”, “modeling”, “traveling”.

GOAL-DTU: Development of Distributed Intelligence for the Multi-Agent Programming Contest

Changes:

- The 45 small figures in Section 2.8 have been moved to Appendix A and the captions have been shortened.

Towards Verifying a Blocks World for Teams GOAL Agent

Changes:

- The published article has a reference "Appendix A" to online material written by myself but in order to make this thesis self-contained the online material has been included in the thesis as Appendix B and the reference updated.

On Using Theorem Proving for Cognitive Agent-Oriented Programming

No additional changes.

Machine-Checked Verification of Cognitive Agents

No additional changes.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Natasha Alechina, Mehdi Dastani, Anas F. Khan, Brian Logan, and John-Jules Meyer. Using Theorem Proving to Verify Properties of Agent Programs. In *Specification and Verification of Multi-agent Systems*, pages 1–33. Springer, 2010.

[3] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem Proving in Lean, 2020. https://leanprover.github.io/theorem_proving_in_lean/.

[4] Tristan Behrens, Koen Hindriks, Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Jürgen Dix, Jomi Fred Hübner, and Alexander Pokahr. An Interface for Agent-Environment Interaction. In Rem W. Collier, Jürgen Dix, and Peter Novák, editors, *Programming Multi-Agent Systems - 8th International Workshop, ProMAS 2010*, volume 6599 of *Lecture Notes in Computer Science*, pages 139–158. Springer, 2010.

[5] Tristan Behrens, Koen Hindriks, and Jürgen Dix. Towards an Environment Interface Standard for Agent Platforms. *Ann. Math. Artif. Intell.*, 61:261–295, 2011.

[6] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE: A FIPA2000 Compliant Agent Development Environment. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 216–217. Association for Computing Machinery, 2001.

[7] Yves Bertot and Pierre Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[8] Olivier Boissier, Rafael H. Bordini, Jomi Hubner, and Alessandro Ricci. *Programming Multi-Agent Systems Using JaCaMo*. MIT Press, 2020.

[9] Rafael Bordini, Michael Fisher, Michael Wooldridge, and Willem Visser. Model Checking Rational Agents. *Intelligent Systems, IEEE*, 19:46–52, 2004.

[10] Rafael Bordini, Jomi Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley, 2007.

[11] Rafael H. Bordini, Amal El Fallah Seghrouchni, Koen Hindriks, Brian Logan, and Alessandro Ricci. Agent Programming in the Cognitive Era. *Autonomous Agents and Multi-Agent Systems*, 34(2):1–31, 2020.

[12] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying Multi-agent Programs by Model Checking. *Autonomous Agents and Multi-Agent Systems*, 12:239–256, 2006.

[13] Niklas Skamriis Boss, Andreas Schmidt Jensen, and Jørgen Villadsen. Building Multi-Agent Systems Using Jason. *Annals of Mathematics and Artificial Intelligence*, 59(3-4):373–388, 2010.

[14] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4:349–355, 1988.

[15] Nils Bulling and Koen V. Hindriks. Communicating Rational Agents: Semantics and Verification. Technical report, Clausthal University of Technology, Clausthal, Germany, 2009.

[16] Nils Bulling and Koen V. Hindriks. Towards a Verification Framework for Communicating Rational Agents. *MATES*, pages 177–182, 2009.

[17] Roberta Calegari, Giovanni Ciatto, Viviana Mascardi, and Andrea Omicini. Logic-Based Technologies for Multi-Agent Systems: A Systematic Literature Review. *Autonomous Agents and Multi-Agent Systems*, 35, 2020.

[18] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19:7–34, 2001.

[19] Philip Cohen and Hector Levesque. Intention = Choice + Commitment. In *Proceedings of AAAI-87*, volume 42, pages 410–415, 1987.

[20] Rem W. Collier, Seán Russell, and David Lillis. Exploring AOP from an OOP Perspective. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 25–36. Association for Computing Machinery, 2015.

[21] Mehdi Dastani, Jaap Brandsema, Amco Dubel, and John-Jules Meyer. Debugging BDI-Based Multi-Agent Programs. In *Proceedings of the 7th International Conference on Programming Multi-Agent Systems*, ProMAS'09, pages 151–169, 2010.

[22] Frank S. de Boer, Koen Hindriks, Wiebe van der Hoek, and John-Jules Meyer. A Verification Framework for Agent Programming with Declarative Goals. *Journal of Applied Logic*, 5:277–302, 2007.

[23] Louise Dennis, Michael Fisher, Matthew P. Webster, and Rafael Heitor Bordini. Model Checking Agent Programming Languages. *Automated Software Engineering*, 19:5–63, 2012.

[24] Louise A. Dennis and Michael Fisher. Programming Verifiable Heterogeneous Agent Systems. In *Programming Multi-Agent Systems*, pages 40–55. Springer, 2009.

[25] The GOAL Developers. The GOAL Agent Programming Language, 2022. https://goalapl.atlassian.net/wiki/spaces/GOAL/.

[26] The Isabelle Developers. The Isabelle Proof Assistant, 2022. https://isabelle.in.tum.de/.

[27] Hans van Ditmarsch, Wiebe van der Hoek, and Barteld Kooi. *Dynamic Epistemic Logic*. Studies In Epistemology, Logic, Methodology and Philosophy Of Science. Springer, Dordrecht, 2007.

[28] Jürgen Dix, Brian Logan, and Michael Winikoff. Engineering Reliable Multiagent Systems (Dagstuhl Seminar 19112). *Dagstuhl Reports*, 9(3):52–63, 2019.

[29] Mikko Berggren Ettienne, Steen Vester, and Jørgen Villadsen. Implementing a Multi-Agent System in Python with an Auction-Based Agreement Approach. In *Programming Multi-Agent Systems*, Lecture Notes in Artificial Intelligence, pages 185–196. Springer, 2012. 9th International Workshop on Programming Multi-Agent Systems (ProMAS 2011).

[30] Andreas Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. Teaching a Formalized Logical Calculus. *Electronic Proceedings in Theoretical Computer Science*, 313:73–92, 2020. 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu'19.

[31] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[32] John Harrison. HOL Light: A Tutorial Introduction. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, pages 265–269. Springer Berlin Heidelberg, 1996.

[33] Koen Hindriks. *Agent Programming Languages: Programming with Mental Models*. PhD thesis, Utrecht University, 2001. https://dspace.library.uu.nl/handle/1874/859.

[34] Koen Hindriks. Modules as Policy-Based Intentions: Modular Agent Programming in GOAL. In Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff, editors, *Programming Multi-Agent Systems*, pages 156–171. Springer Berlin Heidelberg, 2008.

[35] Koen Hindriks, Frank de Boer, Wiebe Hoek, and John-Jules Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2:357–401, 1999.

[36] Koen Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Meyer. Agent Programming with Declarative Goals. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 228–243. Springer, 2001.

[37] Koen Hindriks and Jürgen Dix. GOAL: A Multi-Agent Programming Language Applied to an Exploration Game. In *Agent-oriented software engineering*, pages 235–258. Springer, 2014.

[38] Koen Hindriks and Wiebe van der Hoek. GOAL Agents Instantiate Intention Logic. In *Programming Multi-Agent Systems*, pages 196–219, 2008.

[39] Koen V. Hindriks. Programming Rational Agents in GOAL. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009.

[40] Java Pathfinder. https://github.com/javapathfinder/jpf-core/wiki, 2021. Accessed: 2021-11-22.

[41] Alexander Birch Jensen. A Theorem Proving Approach to Formal Verification of a Cognitive Agent. In Kenji Matsui, Sigeru Omatu, Tan Yigitcanlar, and Sara Rodriguez-González, editors, *Distributed Computing and Artificial Intelligence, Volume 1: 18th International Conference, DCAI 2021, Salamanca, Spain, 6-8 October 2021*, volume 327 of *Lecture Notes in Networks and Systems*, pages 1–11. Springer, 2021.

[42] Alexander Birch Jensen. Formal Verification of a Cognitive Agent Using Theorem Proving. In *9th International Workshop on Engineering Multi-Agent Systems (EMAS 2021)*, 2021.

[43] Alexander Birch Jensen. Towards Verifying a Blocks World for Teams GOAL Agent. In Ana Paula Rocha, Luc Steels, and Jaap van den Herik, editors, *Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021)*, volume 1, pages 337–344. Science and Technology Publishing, 2021.

[44] Alexander Birch Jensen. Towards Verifying GOAL Agents in Isabelle/HOL. In Ana Paula Rocha, Luc Steels, and Jaap van den Herik, editors, *Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021)*, volume 1, pages 345–352. Science and Technology Publishing, 2021.

[45] Alexander Birch Jensen. Formalization of GOAL in Isabelle, 2022. https://people.compute.dtu.dk/aleje/#public.

[46] Alexander Birch Jensen. Machine-Checked Verification of Cognitive Agents. In Ana Paula Rocha, Luc Steels, and Jaap van den Herik, editors, *Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022)*, volume 1, pages 245–256. Science and Technology Publishing, 2022.

[47] Alexander Birch Jensen, Koen V. Hindriks, and Jørgen Villadsen. On Using Theorem Proving for Cognitive Agent-Oriented Programming. In Ana Paula Rocha, Luc Steels, and Jaap van den Herik, editors, *Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021)*, volume 1, pages 446–453. Science and Technology Publishing, 2021.

[48] Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. First-Order Logic According to Harrison. *Archive of Formal Proofs*, 2017. https://isa-afp.org/entries/FOL_Harrison.html, Formal proof development.

[49] Alexander Birch Jensen and Jørgen Villadsen. GOAL-DTU: Development of Distributed Intelligence for the Multi-Agent Programming Contest. In Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, and Tabajara Krausburg, editors, *The Multi-Agent Programming Contest 2019*, Lecture Notes in Computer Science, pages 79–105. Springer, 2020. 14th Annual Multi-Agent Programming Contest, MAPC 2019.

[50] Alexander Birch Jensen, Jørgen Villadsen, Jonas Weile, and Erik Kristian Gylling. The 15th Edition of the Multi-Agent Programming Contest - The GOAL-DTU Team. In Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, and Tabajara Krausburg, editors, *The Multi-Agent Programming Contest 2021*, pages 46–81. Springer, 2021.

[51] Matthew Johnson, Catholijn Jonker, Birna Riemsdijk, Paul J. Feltovich, and Jeffrey Bradshaw. Joint Activity Testbed: Blocks World for Teams (BW4T). *ESAW*, pages 254–256, 2009.

[52] Sung-Shik Jongmans, Koen Hindriks, and M. Riemsdijk. Model Checking Agent Programs by Using the Program Interpreter. In *CLIMA*, pages 219–237, 2010.

[53] Vincent Koeman, Koen Hindriks, and Catholijn Jonker. Omniscient Debugging for Cognitive Agent Programs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 265–272, 2017.

[54] Vincent Koeman, Koen Hindriks, and Catholijn Jonker. Automating Failure Detection in Cognitive Agent Programs. *International Journal of Agent-Oriented Software Engineering*, 6:275–308, 2018.

[55] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *ACM Comput. Surv.*, 52(5), 2019.

[56] Jayadev Misra. A Logic for Concurrent Programming. Technical report, Formal Aspects of Computing, 1994.

[57] Nils J. Nilsson. Probabilistic Logic. *Artificial Intelligence*, 28(1):71–87, 1986.

[58] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[59] The Multi-Agent Programming Contest Organizers. The Multi-Agent Programming Contest, 2022. https://multiagentcontest.org/.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

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

[62] Lawrence Paulson. Computational Logic: Its Origins and Applications. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 474, 2017.

[63] Lawrence Paulson, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *Formal Aspects Comput.*, 31(6):675–698, 2019.

[64] Lawrence C. Paulson. Formalising Mathematics in Simple Type Theory. In Stefania Centrone, Deborah Kant, and Deniz Sarikaya, editors, *Reflections on the Foundations of Mathematics: Univalent Foundations, Set Theory and General Thoughts*, pages 437–453. Springer, 2019.

[65] Stefan Poslad. Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems*, 2(4):15–39, 2007.

[66] Anand S. Rao. AgentSpeak(L): BDI Agents Speak out in a Logical Computable Language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Agents Breaking Away: Agents Breaking Away*, MAAMAW '96, pages 42–55. Springer-Verlag, 1996.

[67] Anand S. Rao and Michael P. Georgeff. 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., 1991.

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

[69] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends in Programming Languages*, 5(2-3):102–281, 2019.

[70] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[71] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. The Cognitive Agents Specification Language and Verification Environment for Multiagent Systems. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1*, AAMAS '02, pages 19–26. Association for Computing Machinery, 2002.

[72] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[73] Konrad Slind and Michael Norrish. HOL-4 manuals, 1998-2020. https://hol-theorem-prover.org/.

[74] Sonja Smets and Anthia Solaki. The Effort of Reasoning: Modelling the Inference Steps of Boundedly Rational Agents. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10944:307–324, 2018.

[75] The Agda Developers. The Agda Wiki, 2020. https://wiki.portal.chalmers.se/agda/pmwiki.php.

[76] Alasdair Urquhart. Basic Many-Valued Logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 2, pages 249–295. Springer, 2001.

[77] Steen Vester, Niklas Skamriis Boss, Andreas Schmidt Jensen, and Jørgen Villadsen. Improving Multi-Agent Systems Using Jason. *Annals of Mathematics and Artificial Intelligence*, 61(4):297–307, 2011.

[78] Jørgen Villadsen, Oliver Fleckenstein, Helge Hatteland, and John Bruntse Larsen. Engineering a Multi-Agent System in Jason and CArtAgO. *Annals of Mathematics and Artificial Intelligence*, 84(1-2):57–174, 2018.

[79] Jørgen Villadsen, Andreas Halkjær From, Salvador Jacobi, and Nikolaj Nøkkentved Larsen. Multi-Agent Programming Contest 2016 — The Python-DTU Team. *International Journal of Agent-Oriented Software Engineering*, 6(1):86–100, 2018.

[80] Jørgen Villadsen, Asta Halkjær From, Alexander Birch Jensen, and Anders Schlichtkrull. Interactive Theorem Proving for Logic and Information. In Roussanka Loukanova, editor, *Natural Language Processing in Artificial Intelligence — NLPinAI 2021*, pages 25–48. Springer, 2022.

[81] Jørgen Villadsen, Alexander Birch Jensen, and Anders Schlichtkrull. NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle. *IfCoLog Journal of Logics and their Applications*, 4(1):55–82, 2017.

[82] Jørgen Villadsen, Andreas Schmidt Jensen, Nicolai Christian Christensen, Andreas Viktor Hess, Jannick Boese Johnsen, Øyvind Grønland Woller, and Philip Bratt Ørum. Engineering a Multi-Agent System in GOAL. In *Engineering Multi-Agent Systems*, Lecture Notes in Computer Science, pages 329–338. Springer, 2013. 1st International Workshop on Engineering Multi-Agent Systems (EMAS 2013).

[83] Jørgen Villadsen, Andreas Schmidt Jensen, Mikko Berggren Ettienne, Steen Vester, Kenneth Balsiger Andersen, and Andreas Frøsig. Reimplementing a Multi-Agent System in Python. In Mehdi Dastani, Jomi F. Hübner, and Brian Logan, editors, *Programming Multi-Agent Systems*, Lecture Notes in Computer Science, pages 205–216. Springer, 2013. 10th International Workshop on Programming Multi-Agent Systems (ProMAS 2012).

[84] M. Winikoff. Assurance of Agent Systems: What Role Should Formal Verification Play? In Mehdi Dastani, Koen V. Hindriks, and John-Jules Charles Meyer, editors, *Specification and Verification of Multi-agent Systems*, pages 353–383. Springer US, 2010.

[85] Michael Winikoff and Stephen Cranefield. On the Testability of BDI Agent Systems. *Journal of Artificial Intelligence Research*, 51:71–131, 2014.

[86] Lotfi A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965.