

On using Theorem Proving for Cognitive Agent-oriented Programming

Alexander Birch Jensen¹^a, Koen V. Hindriks²^b and Jørgen Villadsen¹^c

¹*DTU Compute, Department of Applied Mathematics and Computer Science, Technical University of Denmark, Richard Petersens Plads, Building 324, DK-2800 Kongens Lyngby, Denmark*

²*Social AI Group, Boelelaan 1111, Vrije Universiteit (VU) Amsterdam, 1081 HV Amsterdam, The Netherlands*

Keywords: Agent-oriented Programming, Cognitive Multi-Agent Systems, Software Reliability, Proof Assistants.

Abstract: 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.


1 INTRODUCTION


Reliability is of key importance for software systems in general and multi-agent systems (MAS) and cognitive MAS (CMAS) in particular (Dix et al., 2019). 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. Because 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 behaviour these mechanisms are able to generate. We often observe complex behaviour patterns of agents that exceed those of traditional procedural programs (Winikoff and Crane-field, 2014).


Testing methods for CMAS have been explored by (Koeman et al., 2018; Dastani et al., 2010), 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 (Hindriks et al., 1999; Hindriks et al., 2001), and AgentSpeak(L) languages such as Jason and JaCaMo (Rao, 1996; Bordini et al., 2007; Boissier et al., 2020), provides a suitable starting point for developing these techniques. The initial developments of BDI logics such as by (Rao and Georgeff, 1991) and (Cohen and Levesque, 1987) did not connect well with the more practically oriented work done in engineering and programming of agents such as (Hindriks et al., 1999). 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 (Jongmans et al., 2010; Bordini et al., 2004) 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 (Ringer et al., 2019). One major branch is based on simple type theory (higher-order logic) with prime examples being HOL Light, HOL4 and Isabelle/HOL (Harrison, 1996; Slind and Norrish, 2020; Nipkow et al., 2002). Another major branch contains those based on dependent type theory such as Agda, Coq and Lean (The Agda Developers, 2020; Bertot and Castéran, 2004; Avigad et al., 2020). Built from a

^a <https://orcid.org/0000-0002-7298-2133>

^b <https://orcid.org/0000-0002-5707-5236>

^c <https://orcid.org/0000-0003-3624-1159>

small, trusted kernel proof assistants provide tools for ensuring reliability of systems that are also trustworthy (Paulson et al., 2019). 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 2 highlights some of the relevant existing work in the literature. Section 3 gives a brief introduction to the basic concepts of the programming language GOAL. Section 4 discusses the challenge of ensuring reliability of CMAS and compares the current available approaches for demonstrating reliability. Section 5 explores how a theorem proving approach can contribute to demonstrating reliability. Section 6 serves as an appetizer for early work on formalizing GOAL in Isabelle/HOL. Finally, Section 7 makes some concluding remarks.

2 RELATED WORK

Agent systems, especially in the multi-agent setting, are notoriously hard to test. The findings of (Winikoff and Cranefield, 2014) 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 straight-forward statement that the agent will eventually succeed is too strong and not provable. (Winikoff, 2010) 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: (Jongmans et al., 2010) 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. (Koeman et al., 2018) 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.

(Koeman et al., 2017) 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 (Alechina et al., 2010) and in particular by (Shapiro et al., 2002) 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 (Jensen, 2021b) where the formalization of GOAL in

Isabelle/HOL (to be presented in Section 6) is discussed in details; (Jensen, 2021a) 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.

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 (Hindriks et al., 2001; Hindriks, 2009; Hindriks and Dix, 2014).

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 (Misra, 1994). 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 (Rao and Georgeff, 1993):

- Σ is consistent, i.e. does not 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 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 behaviour 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.

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. (Calegari et al., 2020) 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 (Clarke et al., 2001). 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 (Hindriks and van der Hoek, 2008).

Some of the typical challenges for engineering CMAS reliably are: knowledge representation (KR),

Table 1: Programming concepts of multi-agent systems and possible approaches for demonstrating their reliability.

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

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 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 (★).

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.

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) (Paulson, 2017). 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 (Paulson, 2019).

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?

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 (de Boer et al., 2007).

In the following we will illustrate our suggested theorem proving approach using Isabelle/HOL. For further details see (Jensen, 2021b).

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 (Jensen, 2021a).

In Isabelle/HOL expressions to be parsed within the context of the (embedded) higher-order language are encapsulated in the so-called “cartouches” $\langle \dots \rangle$. Note that they generally can be omitted for atomic expressions.

Cognitive concepts of agents are modelled 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 $mst = \langle (\Phi_L \text{ set} \times \Phi_L \text{ set}) \rangle$

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* :: $\langle mst \Rightarrow bool \rangle (\nabla)$ **where**
 $\langle \nabla x \equiv let (\Sigma, \Gamma) = x \text{ in}$
 $\Sigma \Vdash_C \perp_L \wedge (\forall \gamma \in \Gamma. \Sigma \Vdash_C \gamma \wedge \{ \} \Vdash_C \neg_L \gamma) \rangle$

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 $\langle mst \Rightarrow bool \rangle$: 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 $let (\Sigma, \Gamma) = x \text{ in } \dots$ introduces local variables Σ and Γ in \dots 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 $\Phi_M =$
 $B \Phi_L \mid$
 $G \Phi_L \mid$
 $Neg \Phi_M (\neg_M) \mid$
 $Imp \Phi_M \Phi_M (\mathbf{infix} \ \langle \longrightarrow_M \rangle \ 60) \mid$
 $Dis \Phi_M \Phi_M (\mathbf{infixl} \ \langle \vee_M \rangle \ 70) \mid$
 $Con \Phi_M \Phi_M (\mathbf{infixl} \ \langle \wedge_M \rangle \ 80)$

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 \mid 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* :: $\langle mst \Rightarrow \Phi_M \Rightarrow bool \rangle (\mathbf{infix} \ \langle \models_M \rangle \ 50)$ **where**

$\langle M \models_M (B \Phi) = (let (\Sigma, -) = M \text{ in } \Sigma \Vdash_C \Phi) \mid$
 $\langle M \models_M (G \Phi) = (let (\Sigma, \Gamma) = M \text{ in}$
 $\Sigma \Vdash_C \Phi \wedge (\exists \gamma \in \Gamma. \{ \} \Vdash_C \gamma \longrightarrow_L \Phi)) \mid$
 $\langle M \models_M (\neg_M \Phi) = (\neg M \models_M \Phi) \mid$
 $\langle M \models_M (\Phi_1 \longrightarrow_M \Phi_2) = (M \models_M \Phi_1 \longrightarrow M \models_M \Phi_2) \mid$
 $\langle M \models_M (\Phi_1 \vee_M \Phi_2) = (M \models_M \Phi_1 \vee M \models_M \Phi_2) \mid$
 $\langle M \models_M (\Phi_1 \wedge_M \Phi_2) = (M \models_M \Phi_1 \wedge M \models_M \Phi_2) \rangle$

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

As a slight deviation from (de Boer et al., 2007), 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

$$\langle \neg (\forall \Sigma \Gamma \Phi \Psi. \nabla (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G (\Phi \longrightarrow_L \Psi)) \longrightarrow_M G \Phi \longrightarrow_M G \Psi \rangle$$

and

$$\langle \neg (\forall \Sigma \Gamma \Phi \Psi. \nabla (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G (\Phi \wedge_L (\Phi \longrightarrow_L \Psi)) \longrightarrow_M G \Psi \rangle$$

and

$$\langle \neg (\forall \Sigma \Gamma \Phi \Psi. \nabla (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G \Phi \wedge_M G \Psi \longrightarrow_M G (\Phi \wedge_L \Psi)) \rangle$$

and

$$\langle \{ \} \models_C \Phi \longleftarrow_L \Psi \longrightarrow \nabla (\Sigma, \Gamma) \longrightarrow (\Sigma, \Gamma) \models_M G \Phi \longleftarrow_M G \Psi \rangle$$

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 $derive :: \langle \Phi_M \Rightarrow bool \rangle (\vdash_M)$ **where**

$$R1: \langle conv (map-of T) \varphi' = \varphi \Longrightarrow \{ \# \} \vdash_C \varphi' \Longrightarrow \vdash_M \varphi \rangle |$$

$$R2: \langle \{ \# \} \vdash_C \Phi \Longrightarrow \vdash_M (B \Phi) \rangle |$$

$$A1: \langle \vdash_M (B (\Phi \longrightarrow_L \Psi)) \longrightarrow_M (B \Phi \longrightarrow_M B \Psi) \rangle |$$

$$A2: \langle \vdash_M (\neg_M (B \perp_L)) \rangle |$$

$$A3: \langle \vdash_M (\neg_M (G \perp_L)) \rangle |$$

$$A4: \langle \vdash_M ((B \Phi) \longrightarrow_M (\neg_M (G \Phi))) \rangle |$$

$$A5: \langle \{ \# \} \vdash_C (\Phi \longrightarrow_L \Psi) \Longrightarrow \vdash_M (\neg_M (B \Psi) \longrightarrow_M (G \Phi)) \longrightarrow_M (G \Psi) \rangle$$

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 $\langle \nabla M \rangle$

shows $\langle \vdash_M \Phi \Longrightarrow M \models_M \Phi \rangle$

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.

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.

REFERENCES

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

- Avigad, J., Moura, L. d., and Kong, S. (2020). Theorem Proving in Lean. <https://leanprover.github.io/theorem-proving-in-lean/>.
- Bertot, Y. and Castéran, P. (2004). *Coq'Art: The Calculus of Inductive Constructions*. Springer.
- Boissier, O., Bordini, R. H., Hubner, J., and Ricci, A. (2020). *Programming Multi-Agent Systems Using Ja-CaMo*. MIT Press.
- Bordini, R., Fisher, M., Wooldridge, M., and Visser, W. (2004). Model Checking Rational Agents. *Intelligent Systems, IEEE*, 19:46–52.
- Bordini, R., Hübner, J., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley.
- Calegari, R., Ciatto, G., Mascardi, V., and Omicini, A. (2020). Logic-based technologies for multi-agent systems: a systematic literature review. *Autonomous Agents and Multi-Agent Systems*, 35.
- Clarke, E., Biere, A., Raimi, R., and Zhu, Y. (2001). Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19:7–34.
- Cohen, P. and Levesque, H. (1987). Intention = Choice + Commitment. In *Proceedings of AAAI-87*, volume 42, pages 410–415.
- Dastani, M., Brandsema, J., Dubel, A., and Meyer, J.-J. (2010). Debugging BDI-Based Multi-Agent Programs. In *Programming Multi-Agent Systems*, pages 151–169.
- de Boer, F. S., Hindriks, K. V., van der Hoek, W., and Meyer, J.-J. (2007). A verification framework for agent programming with declarative goals. *Journal of Applied Logic*, 5:277–302.
- Dix, J., Logan, B., and Winikoff, M. (2019). Engineering Reliable Multiagent Systems (Dagstuhl Seminar 19112). *Dagstuhl Reports*, 9(3):52–63.
- Harrison, J. (1996). HOL Light: A tutorial introduction. In Srivas, M. and Camilleri, A., editors, *Formal Methods in Computer-Aided Design*, pages 265–269, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Hindriks, K. and van der Hoek, W. (2008). GOAL Agents Instantiate Intention Logic. In *Programming Multi-Agent Systems*, pages 196–219.
- Hindriks, K. V. (2009). Programming rational agents in goal. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., and Bordini, R. H., editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, Boston, MA.
- Hindriks, K. V., Boer, F., van der Hoek, W., and Meyer, J.-J. (1999). Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2:357–401.
- Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. (2001). Agent Programming with Declarative Goals. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 228–243. Springer.
- Hindriks, K. V. and Dix, J. (2014). GOAL: A Multi-agent Programming Language Applied to an Exploration Game. In *Agent-oriented software engineering*, pages 235–258. Springer.
- Jensen, A. B. (2021a). Towards Verifying a Blocks World for Teams GOAL Agent. In *ICAART 2021 - Proceedings of the 13th International Conference on Agents and Artificial Intelligence*. SciTePress. To appear.
- Jensen, A. B. (2021b). Towards Verifying GOAL Agents in Isabelle/HOL. In *ICAART 2021 - Proceedings of the 13th International Conference on Agents and Artificial Intelligence*. SciTePress. To appear.
- Jongmans, S.-S., Hindriks, K., and Riemdsdijk, M. (2010). Model Checking Agent Programs by Using the Program Interpreter. In *CLIMA*, pages 219–237.
- Koeman, V., Hindriks, K., and Jonker, C. (2017). Omniscient debugging for cognitive agent programs. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 265–272.
- Koeman, V., Hindriks, K., and Jonker, C. (2018). Automating failure detection in cognitive agent programs. *International Journal of Agent-Oriented Software Engineering*, 6:275–308.
- Misra, J. (1994). A Logic for Concurrent Programming. Technical report, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas, USA.
- Nipkow, T., Paulson, L., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer.
- Paulson, L. (2017). Computational logic: Its origins and applications. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 474.
- Paulson, L., Nipkow, T., and Wenzel, M. (2019). From LCF to Isabelle/HOL. *Formal Aspects Comput.*, 31(6):675–698.
- Paulson, L. C. (2019). Formalising mathematics in simple type theory. In Centrone, S., Kant, D., and Sarikaya, D., editors, *Reflections on the Foundations of Mathematics: Univalent Foundations, Set Theory and General Thoughts*, pages 437–453. Springer International Publishing, Cham.
- Rao, A. S. (1996). 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, Berlin, Heidelberg. Springer-Verlag.
- Rao, A. S. and Georgeff, M. P. (1991). Modeling Rational Agents within a BDI-Architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, KR'91*, pages 473–484. Morgan Kaufmann Publishers Inc.
- Rao, A. S. and Georgeff, M. P. (1993). Intentions and Rational Commitment. In *Proceedings of the First Pacific Rim Conference on Artificial Intelligence (PRICAI-90)*. Citeseer.
- Ringer, T., Palmkog, K., Sergey, I., Gligoric, M., and Tatlock, Z. (2019). QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281.

- Shapiro, S., Lespérance, Y., and Levesque, H. J. (2002). 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, New York, NY, USA. Association for Computing Machinery.
- Slind, K. and Norrish, M. (1998-2020). HOL-4 manuals. <https://hol-theorem-prover.org/>.
- The Agda Developers (2020). The Agda Wiki. <https://wiki.portal.chalmers.se/agda/pmwiki.php>.
- Winikoff, M. (2010). Assurance of Agent Systems: What Role Should Formal Verification Play? In Dastani, M., Hindriks, K. V., and Meyer, J.-J. C., editors, *Specification and Verification of Multi-agent Systems*, pages 353–383. Springer US, Boston, MA.
- Winikoff, M. and Cranefield, S. (2014). On the Testability of BDI Agent Systems. *Journal of Artificial Intelligence Research*, 51:71–131.

