TU Delft

Delft University of Technology

Tools for Developing Cognitive Agents

Koeman, Vincent

DOI
10.4233/uuid:f80750ee-db68-480e-8c58-2c167bd24ee5

Publication date
2019

Document Version
Final published version

Citation (APA)
Koeman, V. (2019). *Tools for Developing Cognitive Agents*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:f80750ee-db68-480e-8c58-2c167bd24ee5

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Tools for Developing Cognitive Agents

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 18 juni 2019 om 15:00 uur

door

Vincent Jaco KOEMAN

Master of Science in Computer Science,
Technische Universiteit Delft, Nederland,
geboren te Hoorn, Nederland.

Dit proefschrift is goedgekeurd door de promotoren.

Samenstelling promotiecommissie bestaat uit:

Rector Magnificus	voorzitter
Prof. dr. K.V. Hindriks	Technische Universiteit Delft / VU Amsterdam, promotor
Prof. dr. C.M. Jonker	Technische Universiteit Delft, promotor

Onafhankelijke leden:

Prof. dr. A. El Fallah Seghrouchni	Sorbonne University, Frankrijk
Prof. dr. M.M. Dastani	Universiteit Utrecht
Prof. dr. T. Holvoet	Katholieke Universiteit Leuven, België
Prof. dr. F.M. Brazier	Technische Universiteit Delft
Prof. dr. ir. M.J.T. Reinders	Technische Universiteit Delft, reservelid

Printed by: Haveka

Front & Back: W.J. Buijs

Copyright © 2019, V.J. Koeman. All rights reserved.

SIKS Dissertation Series No. 2019-19
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-94-6366-167-6

An electronic version of this dissertation is available at
https://repository.tudelft.nl

Contents

Summary **v**

Samenvatting **vii**

1 Introduction **1**
1.1 Developing Cognitive Agents 2
1.2 Research Questions 3
1.3 Approach 4
References. 6

2 Designing a Source-Level Debugger for Cognitive Agents **9**
2.1 Introduction 10
2.2 Issues in Debugging Cognitive Agent Programs 10
2.3 Debugger Design Approach 21
2.4 Evaluation 35
2.5 Conclusions and Future Work 39
References. 40

3 Automating Failure Detection in Cognitive Agents **45**
3.1 Introduction 46
3.2 Related Work 47
3.3 Automated Testing Framework. 48
3.4 Testing for Failures. 54
3.5 Testing GOAL Agents in the Eclipse IDE 64
3.6 Evaluation 65
3.7 Conclusions and Future Work 74
References. 75

4 Facilitating Omniscient Debugging for Cognitive Agents **79**
4.1 Introduction 80
4.2 Related Work 80
4.3 Agent Trace Design. 84
4.4 Evaluation 88
4.5 Visualizing Traces 90
4.6 Conclusions and Future Work 92
References. 92

5 Designing a Cognitive Connector for Complex Environments **95**
5.1 Introduction 96
5.2 Related Work 97
5.3 Case Study: StarCraft 98

5.4 Connector Design Approach 99
5.5 Conclusions and Future Work 111
References 112

6 Conclusion **115**
6.1 Conclusions 115
6.2 Limitations 117
6.3 Contributions 118
6.4 Future Work. 119
References 121

A Source-Level Debugger Questionnaire and Correlation Analysis **123**

Curriculum Vitæ **125**

List of Publications **127**

SIKS Dissertation Series **129**

Acknowledgements **143**

Summary

Agent-oriented programming (AOP) is a programming paradigm introduced roughly thirty years ago as an approach to problems in Artificial Intelligence (AI). An agent is a piece of software that can perceive its environment (e.g., through sensors) and act upon that environment (e.g., through actuators). A cognitive agent is a specific type of agent that executes a decision cycle in which it processes events and selects actions based on cognitive notions such as beliefs and goals. Often, multiple agents are used, which is referred to as a multi-agent system (MAS). MAS is generally advertised an approach to handling problems that require multiple problem solving methods, multiple perspectives, and multiple problem solving entities.

Tools and techniques for the programming of cognitive agents need to be based on the underlying agent-oriented paradigm, which is a significant challenge, as unlike more traditional paradigms, they should for example take into account that agents execute a specific decision cycle and operate in non-deterministic environments. Therefore, in this thesis, we take existing AOP theories a step further by designing tools for the development of cognitive agent programs with an explicit focus on usability. Each development tool we propose is extensively evaluated on hundreds of (novice) agent programmers. In the context of AOP, the process of detecting, locating and correcting mistakes in a computer program, known as debugging, is particularly challenging. As large part of the effort of a programmer consists of debugging a program, efficient debugging is an essential factor for both productivity and program quality. In this thesis, we contribute both to the process of locating mistakes in agent programs as well as the process of identifying misbehaviour of an agent in the first place.

First, we propose a source-level debugger design for cognitive agents aimed at providing a better insight into the relationship between program code and the resulting behaviours. We identify two different types of breakpoints specifically for agent programming: code-based and cycle-based. The former are based on the structure of an agent program, whereas the latter are based on an agent's decision cycle. We propose design steps for designing a debugger for cognitive agents; by using the syntax and decision cycle of an agent programming language, a set of pre-defined breakpoints and the flow between them can be determined in a structured manner, and represented in a stepping diagram. Based on such a diagram, features such as user-defined breakpoints, visualization of the execution flow, and state inspection can be handled. We provide a design for the GOAL and Jason programming languages, as well as a full implementation for GOAL, and argue that our approach can be applied to other agent programming languages.

Next, we propose an automated testing framework for cognitive agents. We identify a minimal set of temporal operators that enable the specification of test conditions. We show that the resulting test language is sufficiently expressive for

detecting all failure types of an existing taxonomy of failures for cognitive agents. We also introduce an approach for specifying test templates that supports programmers in writing tests for their agents. The proposed test language is minimal in the sense that only two temporal operators are provided. We show by analysing different agent program samples that the language is nevertheless sufficient for detecting failures in cognitive agent programs. An implementation of the proposed framework for the GOAL agent programming language serves as a prototype for evaluation and as an example for other agent programming languages.

Moreover, we show that for AOP back-in-time debugging, a technique that facilitates debugging by moving backwards in time through a program's execution, is possible in practice. We design a tracing mechanism for efficiently storing and exploring agent program runs. We are the first to demonstrate that this mechanism does not affect program runs by empirically establishing that the same tests succeed or fail. This is in stark contrast with previous work in different paradigms, in which the overhead caused by tracing is so large that the technique cannot be effectively used in practice. Usability is supported by a trace visualization method aimed at allowing developers of cognitive agents to more effectively locate mistakes.

Finally, cognitive agents specifically require a connector to their target environment. However, connecting agents with an environment that puts strict real-time constraints on the responsiveness of agents, requires coordination at different levels, and requires complex reasoning about long-term goals under a high level of uncertainty is not a trivial task. In this thesis, we therefore present a design approach for creating connectors for cognitive agent technology to complex environments, illustrated by a case study of such a connector that provides full access to the game StarCraft: Brood War. A major challenge that is addressed is to ensure corresponding cognitive agents can be programmed at a high level of abstraction whilst simultaneously allowing sufficient variety in strategies to be implemented. The viability of the approach is demonstrated by multiple large-scale practical uses of the StarCraft connector, resulting in a varied set of competitive AI systems.

We contribute to the field of developing cognitive agents by empirically investigating the needs of developers of cognitive agents in effectively engineering solutions to AI problems. We introduce design methods for the creation of source-level debuggers, automated testing frameworks, back-in-time debuggers, and cognitive connectors; all vital tools for engineering MAS. Each tool is implemented in the GOAL agent platform, making sure the proposed design approaches are feasible in practice and serving both as a prototype for use in evaluations as well as an open-source example for the developers of other AOP solutions. We believe this work enhances the potential of demonstrating the added value of cognitive agents. First, empowering developers of cognitive agents to effectively debug and test their systems should enhance their potential willingness to employ these technologies. Second, providing developers with a design approach for developing efficient cognitive connectors to complex environments allows AOP to be actually employed for engineering large-scale complex distributed systems. Finally, our empirical results provide concrete examples of the potential of AOP.

Samenvatting

Agentgeoriënteerd programmeren (AOP) is een programmeerparadigma dat ongeveer dertig jaar geleden is geïntroduceerd als een aanpak voor problemen in de Kunstmatige Intelligentie (AI). Een agent is een stuk software dat zijn omgeving kan waarnemen (bv. met sensoren) en in die omgeving kan acteren (bv. door te bewegen). Een cognitieve agent is een specifieke soort agent die een beslissingscyclus uitvoert waarin gebeurtenissen worden verwerkt en acties worden geselecteerd op basis van cognitieve noties als kennis en doelen. Vaak worden meerdere agenten gebruikt, wat een multi-agent systeem (MAS) wordt genoemd. MAS wordt over het algemeen geadverteerd als een aanpak voor problemen die meerdere probleemoplossingsmethoden, meerdere perspectieven, en meerdere probleemoplossende entiteiten nodig hebben.

Hulpmiddelen en technieken voor het programmeren van cognitieve agenten moeten gebaseerd zijn op het onderliggende agentgeoriënteerde paradigma, wat een grote uitdaging is in verhouding met traditionelere paradigma doordat agenten een specifieke beslissingscyclus uitvoeren en in niet-deterministische omgevingen opereren. In dit proefschrift gaan we daarom een stap verder met bestaande AOP-theorieën door hulpmiddelen voor het ontwikkelen van cognitieve agentprogramma's te ontwerpen met een focus op bruikbaarheid. Elke ontwikkeltool die wij voorstellen is uitgebreid geëvalueerd op honderden (beginnende) agentprogrammeurs. In de context van AOP is met name het proces van het detecteren, lokaliseren en corrigeren van fouten in een computerprogramma ('debugging') een uitdaging. Omdat een groot deel van de inzet van een programmeur bestaat uit het debuggen van programma's is efficiënt debuggen een essentiële factor voor zowel productiviteit als kwaliteit. In dit proefschrift dragen we zowel aan het proces van het opsporen van fouten in agentprogramma's als het proces van het identificeren van verkeerde gedragingen van een agent bij.

Als eerste stellen we een ontwerp van 'source-level debugger' voor cognitieve agenten voor, gericht op het geven van een beter inzicht in de relatie tussen programmacode en de resulterende gedragingen. We identificeren twee verschillende soorten breakpoints die specifiek zijn voor agentprogrammeren: op basis van code en op basis van de cyclus. De eerste zijn gebaseerd op de structuur van een agentprogramma, terwijl de andere zijn gebaseerd op de beslissingscyclus van een agent. We stellen ontwerpstappen voor het ontwerpen van een debugger voor cognitieve agenten voor; door de syntaxis en de beslissingscyclus van een agentprogrammeertaal te gebruiken kan een set van voorgedefinieerde 'breakpoints' en de loop daartussen op een gestructureerde manier worden bepaald en gerepresenteerd in een diagram. Op basis van een dergelijk diagram kunnen functionaliteiten als door de gebruiker bepaalde breakpoints, visualisaties van de executieflow en inspectie van de programmatoestand worden afgehandeld. We bieden ontwerpen voor de

agentprogrammeertalen GOAL en Jason aan, samen met een volledige implementatie voor GOAL, en beargumenteren dat onze aanpak ook op andere agentprogrammeertalen kan worden toegepast.

Vervolgens stellen we een 'framework' voor het automatisch testen van cognitieve agenten voor. We identificeren een minimale set van temporele operatoren voor waarmee test condities kunnen worden gespecificeerd. We laten zien dat de hieruit voortvloeiende testtaal voldoende expressief is voor het detecteren van alle defecten uit een bestaande taxonomie voor defecten van cognitieve agenten. We introduceren ook een aanpak voor het specificeren van testsjablonen die programmeurs ondersteunen bij het schrijven van tests voor agenten. De voorgestelde testtaal bevat slechts twee temporele operatoren. We laten met verschillende voorbeelden van agentprogramma's zien dat de testtaal toereikend is voor het detecteren van defecten in agentprogramma's. Een implementatie van het voorgestelde framework voor de agentprogrammeertaal GOAL dient als een prototype voor evaluaties en als een voorbeeld voor andere agentprogrammeertalen.

We laten verder zien dat 'terug-in-de-tijd' debuggen, een techniek waarbij debuggen wordt gefaciliteerd door terug te gaan in de executie van een programma, in de praktijk mogelijk is met AOP. We ontwerpen een traceringsmechanisme voor het efficiënt opslaan en verkennen van executies van agentprogramma's. We zijn de eerste die laten zien dat dit mechanisme de uitvoering van programma's niet beïnvloedt door empirisch vast te stellen dat dezelfde tests slagen of mislukken. Dit is sterk in contrast met eerder werk in andere paradigma's waarbij de overhead van het traceren zo groot is dat de techniek niet effectief in de praktijk kan worden gebruikt. De bruikbaarheid wordt gefaciliteerd door een visualisatiemethode voor getraceerde executies, gericht op het effectief opsporen van fouten door ontwikkelaars van cognitieve agenten.

Cognitieve agenten hebben specifiek een 'connector' met hun doelomgeving nodig. Het is echter geen triviale taak om agenten te verbinden met een omgeving die strikte realtime responsiveness (het vermogen om gelijk te doen wat wordt gewenst) van de agenten eist, waarbij er op verschillende niveaus gecoördineerd moet worden en waarbij er complexe redeneringen over lange-termijn doelstellingen met een hoog niveau van onzekerheid nodig zijn. In dit proefschrift presenteren we daarom een ontwerpmethode voor het maken van connectoren voor cognitieve agent technologie en complexe omgevingen, geïllustreerd met een case study van zo'n connector met volledige toegang tot het spel StarCraft: Brood War. Een belangrijke uitdaging die wordt aangepakt is het verzekeren dat bijbehorende cognitieve agenten op een hoog niveau van abstractie kunnen worden geprogrammeerd terwijl tegelijkertijd een voldoende variëteit in strategieën geïmplementeerd kunnen worden. De uitvoerbaarheid van de aanpak wordt gedemonstreerd door de grootschalige inzet van de StarCraft connector in meerdere praktijksituaties, wat resulteerde in een gevarieerde set van competitieve AI-systemen.

We leveren een bijdrage aan het veld van het ontwikkelen van cognitieve agenten door de behoeftes van de ontwikkelaars van cognitieve agenten bij het effectief oplossen van AI-problemen empirisch te onderzoeken. We introduceren ontwerpmethodes voor het maken van source-level debuggers, frameworks voor automa-

tische tests, terug-in-de-tijd debuggers en cognitieve connectoren; stuk voor stuk essentiële hulpmiddelen voor het ontwikkelen van MAS. Elke ontwikkeltool is geïmplementeerd in het GOAL-platform voor agenten, waarmee we garanderen dat de voorgestelde ontwerpmethodes in de praktijk uitvoerbaar zijn en welke zowel als prototype voor evaluaties functioneren als 'open-source' voorbeeld voor de ontwikkelaars van andere AOP oplossingen. We geloven dat dit werk het potentieel voor het demonstreren van de toegevoegde waarde van cognitieve agenten versterkt. Ten eerste, door ontwikkelaars van cognitieve agenten in staat te stellen om hun systemen effectief te debuggen en testen zou hun potentiële bereidheid om deze technieken in te zetten moeten verbeteren. Ten tweede, door ontwikkelaars te voorzien van een ontwerpmode voor het ontwikkelen van efficiënte cognitieve connectoren kan AOP daadwerkelijk worden ingezet om complexe gedistribueerde systemen te maken. Als laatste geven onze empirische concrete voorbeelden van de potentie van AOP.

1
Introduction

Nowadays, when a programmer sets out to create a piece of software, he or she can choose from over 700 'non-esoteric' *programming languages* like Java, Python, C++, and many others [1]. Adequately solving a programming problem requires choosing a language that facilitates the use of the right concepts in the context of that problem. Any programming language can be classified into one or more *paradigms*. A programming paradigm is an approach to programming that makes use of a specific set of concepts that is aimed at solving certain kinds of problems [2]. Object-oriented programming and functional programming are examples of well-known programming paradigms.

In this thesis, we focus on one such paradigm: **agent-oriented programming** (AOP). Introduced roughly thirty years ago as a programming paradigm [3], AOP is an approach to problems in Artificial Intelligence (AI) that is centred around the concept of an *agent*. We take the definition of Russell and Norvig [4] here, as also illustrated in Figure 1.1, in that an agent is a piece of software that can perceive its environment (e.g., through sensors) and act upon that environment (e.g., through actuators). An agent generally does not require (constant) human guidance or intervention, and operates in an environment (either real or simulated) in which other processes take place and other agents exist. A specific type of (software) agent is a **cognitive agent** [5]. Cognitive agents execute a decision cycle in which they process events and derive a choice of action from their beliefs and goals (see also in Figure 1.1). The cognitive notions like beliefs and goals, often grouped in a so-called *cognitive state* (also: mental state), stem from the *Belief Desire Intention* (BDI) philosophy of Bratman [6][1].

Often, multiple (co-operative) agents are used to solve problems, which is referred to as a **multi-agent system** (MAS). Multi-agent systems are generally advertised as an approach to handling problems that require multiple problem solving methods, multiple perspectives, and/or multiple problem solving entities [8]. AOP

[1] Although the term 'BDI agent' is also often used in literature, we use 'cognitive agent' here as it represents a more generic type of agent (of which BDI agents are a subtype).

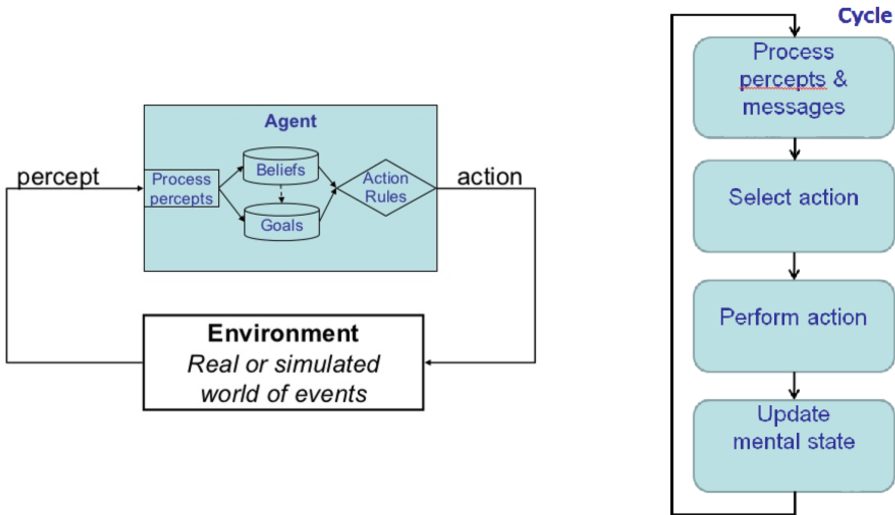


Figure 1.1: A schematic overview of key concepts in (cognitive) AOP is shown on the left; a typical agent decision cycle is shown on the right. Both are from the GOAL agent programming language [7].

offers an alternative to other approaches for engineering complex distributed systems [9, 10]. Applications of MAS are reported in diverse areas such as logistics and manufacturing, telecommunication, aerospace, e-commerce, and defence by Müller and Fischer [11]. An (early) industrial paper [12] even states that "In a wide range of complex business applications ... BDI technology incorporated within an enterprise-level architecture can improve overall developer productivity by an average of 350%." However, currently applications are (still) not widespread, and mostly based on older non-cognitive platforms (as found earlier by Dignum and Dignum [13] as well).

Hindriks [9] argues that the step to mature applications for technologies that support the engineering of cognitive agents is bigger than that of more general purpose frameworks for engineering agents. Cognitive agent technology offers a powerful solution for developing the next generation of agent-based decision-making systems, but as Hindriks [9] also underlines: "it it is time to start paying more attention to the kind of support that a [cognitive] MAS developer needs to facilitate him or her when engineering future MAS applications ... it is important to identify the needs of a developer and **make sure that a developer is provided with the right tools for engineering MAS**."

1.1. Developing Cognitive Agents

Software development is generally performed in an *Integrated Development Environment* (IDE): a set of software tools or applications that provides comprehensive facilities for software development [14]. An IDE consists of tools such as a source

code editor, a compiler or interpreter (or both), build-automation tools, a debugger, and automated testing support [15, 16]. It is, however, challenging to design usable IDEs [17], and perhaps even more so for cognitive agent programming languages, as the paradigm differs largely in many aspects from more traditional paradigms.

Although the need for **dedicated development tools for agent programming** has been broadly recognised for over a decade [9, 18–23], and some methods and tools have been proposed, e.g. [24–29], the current literature has a lack of evaluations performed on agent developers. Tools and techniques for the programming of cognitive agents need to be based on the underlying agent-oriented paradigm [30, 31], which is a significant challenge, as they should, for example, take into account that agents execute a specific decision cycle and operate in non-deterministic environments [32–34]. In this thesis, we take the existing theory a step further by designing tools for the development of cognitive agent programs with an explicit focus on usability. Each development tool we propose has been extensively evaluated on hundreds of (novice) agent programmers.

1.2. Research Questions

The main question that this thesis explores is:

> *How can we support developers of cognitive agents in effectively engineering multi-agent systems?*

In the context of AOP, the literature as mentioned in the previous section specifically indicates that debugging, the process of detecting, locating and correcting faults in a computer program [14], is challenging. A large part of the effort of a programmer consists of debugging a program; this makes efficient debugging an essential factor for both productivity and program quality [35, 36]. A *failure* is an event in which a system does not perform a required function within specified limits [14]. They are caused by a *fault*, an incorrect step, process, or data definition in a program or mistake in a program [14]. Upon detecting a failure, a programmer needs to locate and correct the fault that causes the failure. Tools for debugging thus aim to assist a programmer in detecting failures (i.e., differences between observed and intended behaviour) and locating faults (i.e., find the problem in the code).

Between the two, "fault localization ... is widely recognized to be one of the most tedious, time consuming, and expensive – yet equally critical – activities." [37]. Many fault localization techniques have been developed for paradigms like object-oriented and functional programming, but as detailed in the previous section, the unique approach to programming that is inherent to the AOP paradigm requires a unique approach to its tooling [38]. Moreover, as Wong *et al.* [37] also state, "analyses very often make over-simplified and non-realistic assumptions that do not hold for real-life programs." As we aim to address the specific needs of developers of cognitive agents in this thesis, our first research question is:

> **RQ 1**: *How can we provide developers of cognitive agents with an insight into how observed behaviour relates to the program code?*

Detecting failures in the first place is a major challenge as well [39]. Especially in a multi-agent (and thus concurrent) setting, manually keeping track of the behaviour of all agents is practically infeasible. We thus need to pro-actively detect failures, i.e., automate failure detection. A tool for automated failure detection can even provide clues about the localization of the corresponding fault. Our second research question therefore is:

> **RQ 2**: *How can we automate the detection and localization of failures for developers of cognitive agent programs?*

Even after addressing these two challenges, there are still types of failures in cognitive agent programs for which fault localization is difficult. For example, it is frequently difficult to locate a fault for a failure to execute a certain action based on an agent's current state only; the root cause of a failure in an agent program is more often than not both far removed in time and in code(location). Moreover, real-time programs like multi-agent systems are typically not deterministic. Running the same agent system again more often than not results in a different program run or trace, which further complicates the iterative process of debugging. In order to address these issues, a record (trace) of all decision making processes that took place in an agent's execution (up until the point of failure) is required. Tracing techniques have been developed in different fields (i.e., object-oriented programming). Employing and extending these techniques into the field of cognitive agents, however, is a non-trivial task, resulting in the third research question:

> **RQ 3**: *How can we facilitate developers of cognitive agents in employing 'back-in-time' debugging techniques?*

Finally, in order to address a 'real-world' AI problem, 'just' developing a MAS is not enough. Cognitive agents specifically require a 'connector' to the target environment. However, connecting cognitive agents with an environment that puts strict real-time constraints on the responsiveness of agents, requires coordination at different levels (ranging from a few agents to large groups of agents), and requires complex reasoning about long-term goals under a high level of uncertainty is not a trivial task [40]. Moreover, as such a connector essentially defines which inputs an agent will receive and which outputs it has to decide on, this has a major impact on all aspects of the development of a corresponding MAS as well. Therefore, the fourth and final research question is:

> **RQ 4**: *How can developers of cognitive agents connect their agents to complex real-time environments?*

1.3. Approach

In the following four chapters, each research question is addressed in turn.

In Chapter 2, addressing RQ 1, we propose a design approach for single-step execution (i.e., source-level debugging) of cognitive agents that supports both code-based as well as cycle-based suspension of an agent program. This approach results

in a concrete stepping diagram ready for implementation, as illustrated by a diagram for both the GOAL and Jason agent programming languages, and a corresponding full implementation of a source-level debugger for GOAL in the Eclipse development environment. Based on this implementation, the results of both quantitative and qualitative evaluations on over 200 students are presented and discussed.

In Chapter 3, addressing RQ 2, we propose an automated testing framework for detecting failures in cognitive agent programs. We identify a minimal set of temporal operators that enable the specification of test conditions and show that the test language is sufficiently expressive for detecting all failure types of an existing failure taxonomy for cognitive agents. We also introduce an approach for specifying test templates that supports a programmer in writing tests for cognitive agents. Empirical analysis of agent programs allows us to evaluate whether our approach using test templates adequately detects failures, and to determine the effort that is required to do so in both single and multi agent systems. We also discuss a concrete implementation of the proposed framework for the GOAL agent programming language that has been developed for the Eclipse IDE. Based on this framework, the results of both quantitative and qualitative evaluations on close to 100 pairs of students are presented and discussed.

In Chapter 4, addressing RQ 3, we show that for agent-oriented programming, practical back-in-time ('omniscient') debugging is possible. We design a tracing mechanism for efficiently storing and exploring agent program runs. We are the first to demonstrate that this mechanism does not affect program runs by empirically establishing that the same tests succeed or fail. This is in stark contrast with previous work in different paradigms, in which the overhead caused by tracing is so large that the technique cannot be effectively used in practice. Usability is supported by a trace visualization method aimed at allowing developers of cognitive agents to more effectively locate faults in agent programs. We also discuss a concrete implementation of the proposed tracing mechanism and according visualization for the GOAL agent programming language that has been developed for Eclipse.

In Chapter 5, addressing RQ 4, we propose a design approach to connectors for cognitive agents, based on the principle that each unit that can be controlled in an environment should be mapped onto a single agent. We design and implement a cognitive connector for the real-time strategy (RTS) game StarCraft and use it as a case study for establishing a design method. StarCraft is particularly suitable to this end, as AI for an RTS game such as StarCraft requires the design of complicated strategies for coordinating hundreds of units that need to solve a range of challenges including handling both short-term as well as long-term goals. Our connector is the first implementation that provides full access for cognitive agents to StarCraft: Brood War. We draw several lessons from how our design evolved and from the use of our connector by over 500 students in two years.

Finally, in Chapter 6, we discuss the implication of all these chapters on our main research question. Suggestions for future work are provided based on the results and limitations of our work.

1

References

[1] Wikipedia, *List of programming languages,* https://wikipedia.org/wiki/List_of_programming_languages (2019), accessed: 2019-01-10.

[2] P. Van Roy, *Programming paradigms for dummies: What every programmer should know,* New computational paradigms for computer music **104** (2009).

[3] Y. Shoham, *Agent-oriented programming,* Artificial intelligence **60**, 51 (1993).

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. (Prentice Hall, Upper Saddle River, NJ, USA, 2010).

[5] M. B. van Riemsdijk, *Cognitive agent programming: A semantic approach*, Ph.D. thesis, Utrecht University (2006).

[6] M. Bratman, *Intention, Plans, and Practical Reason* (Center for the Study of Language and Information, 1987).

[7] K. V. Hindriks, *GOAL Programming Guide,* https://bintray.com/artifact/download/goalhub/GOAL/GOALProgrammingGuide.pdf (2018), accessed: 2019-01-10.

[8] N. R. Jennings, K. Sycara, and M. Wooldridge, *A roadmap of agent research and development,* Autonomous Agents and Multi-Agent Systems **1**, 7 (1998).

[9] K. V. Hindriks, *The shaping of the agent-oriented mindset,* in *Engineering Multi-Agent Systems: Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, edited by F. Dalpiaz, J. Dix, and M. B. van Riemsdijk (Springer International Publishing, 2014) pp. 1–14.

[10] B. Logan, *A future for agent programming,* in *Engineering Multi-Agent Systems: Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers*, edited by M. Baldoni, L. Baresi, and M. Dastani (Springer International Publishing, Cham, 2015) pp. 3–17.

[11] J. P. Müller and K. Fischer, *Application impact of multi-agent systems and technologies: A survey,* in *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks*, edited by O. Shehory and A. Sturm (Springer Berlin Heidelberg, 2014) pp. 27–53.

[12] S. S. Benfield, J. Hendrickson, and D. Galanti, *Making a strong business case for multiagent technology,* in *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06 (ACM, New York, NY, USA, 2006) pp. 10–15.

[13] V. Dignum and F. Dignum, *Designing agent systems: State of the practice,* Int. J. Agent-Oriented Softw. Eng. **4**, 224 (2010).

[14] ISO/IEC/IEEE, *24765:2017-9 Systems and software engineering – Vocabulary,* https://www.iso.org/standard/71952.html (2017).

[15] A. Fuggetta, *A classification of CASE technology,* Computer **26**, 25 (1993).

[16] S. P. Reiss, *Software tools and environments,* ACM Computing Surveys **28**, 281 (1996).

[17] R. B. Kline and A. Seffah, *Evaluation of integrated software development environments: Challenges and results from three empirical studies,* International Journal of Human-Computer Studies **63**, 607 (2005).

[18] R. H. Bordini, L. Braubach, J. J. Gomez-Sanz, G. O. Hare, A. Pokahr, and A. Ricci, *A survey of programming languages and platforms for multi-agent systems,* Informatica **30**, 33 (2006).

[19] J. Dix, K. V. Hindriks, B. Logan, and W. Wobcke, *Engineering Multi-Agent Systems (Dagstuhl Seminar 12342),* Dagstuhl Reports **2**, 74 (2012).

[20] M. Winikoff, *Challenges and directions for engineering multi-agent systems,* arXiv preprint arXiv:1209.1428 (2012).

[21] M. B. van Riemsdijk, *20 years of agent-oriented programming in distributed ai: History and outlook,* in *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! 2012 (ACM, New York, NY, USA, 2012) pp. 7–10.

[22] M. Dastani, *A survey of multi-agent programming languages and frameworks,* in *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks*, edited by O. Shehory and A. Sturm (Springer Berlin Heidelberg, 2014) pp. 213–233.

[23] M. Dastani, *Programming multi-agent systems,* The Knowledge Engineering Review **30**, 394 (2015).

[24] D. N. Lam and K. S. Barber, *Debugging agent behavior in an implemented agent system,* in *International Workshop on Programming Multi-Agent Systems* (Springer, 2004) pp. 104–125.

[25] R. Collier, *Debugging agents in Agent Factory,* in *International Workshop on Programming Multi-Agent Systems* (Springer, 2006) pp. 229–248.

[26] J. J. Gomez-Sanz, J. Botía, E. Serrano, and J. Pavón, *Testing and debugging of MAS interactions with INGENIAS,* in *International Workshop on Agent-Oriented Software Engineering* (Springer, 2008) pp. 199–212.

[27] M. Dastani, J. Brandsema, A. Dubel, and J.-J. C. Meyer, *Debugging BDI-based multi-agent programs,* in *International workshop on programming multi-agent systems* (Springer, 2009) pp. 151–169.

[28] Z. Huang, R. Alexander, and J. Clark, *Mutation testing for Jason agents,* in *International Workshop on Engineering Multi-Agent Systems* (Springer, 2014) pp. 309–327.

[29] V. J. Koeman and K. V. Hindriks, *A fully integrated development environment for agent-oriented programming,* in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection*, edited by Y. Demazeau, K. S. Decker, J. Bajo Pérez, and F. de la Prieta (Springer International Publishing, Cham, 2015) pp. 288–291.

[30] Z. Zhang, J. Thangarajah, and L. Padgham, *Model based testing for agent systems,* Software and Data Technologies **22**, 399 (2008).

[31] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah, *Testing in multi-agent systems,* in *Agent-Oriented Software Engineering X*, Vol. 6038 (Springer Berlin Heidelberg, 2011) pp. 180–190.

[32] G. Caire, M. Cossentino, and A. Negri, *Multi-agent systems implementation and testing,* in *Proceedings of the 4th From Agent Theory to Agent Implementation Symposium, AT2AI-4* (2004).

[33] R. Bordini, M. Dastani, and M. Winikoff, *Current issues in multi-agent systems development,* in *Engineering Societies in the Agents World VII*, Lecture Notes in Computer Science, Vol. 4457 (Springer Berlin Heidelberg, 2007) pp. 38–61.

[34] Z. Houhamdi, *Multi-agent system testing: A survey,* International Journal of Advanced Computer Science and Applications **2**, 135 (2011).

[35] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009).

[36] C. Parnin and A. Orso, *Are automated debugging techniques actually helping programmers?* in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11 (ACM, New York, NY, USA, 2011) pp. 199–209.

[37] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, *A survey on software fault localization,* IEEE Transactions on Software Engineering **42**, 707 (2016).

[38] K. Potiron, A. E. F. Seghrouchni, and P. Taillibert, *From fault classification to fault tolerance for multi-agent systems*, SpringerBriefs in Computer Science (Springer, 2013).

[39] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling, *What do we know about defect detection methods?* Software, IEEE **23**, 82 (2006).

[40] D. Weyns, M. Schumacher, A. Ricci, M. Viroli, and T. Holvoet, *Environments in multiagent systems,* Knowledge Engineering Review **20**, 127 (2005).

2

Designing a Source-Level Debugger for Cognitive Agents

When an agent program exhibits unexpected behaviour, a developer needs to locate the fault by debugging the agent's source code. The process of fault localisation requires an understanding of how code relates to the observed agent behaviour. The main aim in this chapter is to design a source-level debugger that supports single-step execution of a cognitive agent program. Cognitive agents execute a decision cycle in which they process events and derive a choice of action from their beliefs and goals. Current state-of-the-art debuggers for agent programs provide insight in how agent behaviour originates from this cycle but less so in how it relates to the program code. As relating source code to generated behaviour is an important part of the debugging task, arguably, a developer also needs to be able to suspend an agent program on code locations.

We propose a design approach for single-step execution of agent programs that supports both code-based as well as cycle-based suspension of an agent program. This approach results in a concrete stepping diagram ready for implementation and is illustrated by a diagram for both the GOAL *and Jason agent programming languages, and a corresponding full implementation of a source-level debugger for* GOAL *in the Eclipse development environment. The evaluation that was performed based on this implementation shows that agent programmers prefer a source-level debugger over a purely cycle-based debugger.*

This chapter has been published in the the Journal of Autonomous Agents and Multi-Agent Systems **31**(5) (2016) [1], an extension of work published in the Conference on Principles and Practices of Multi-Agent Systems (2015) [2].

2.1. Introduction

Debugging is the process of detecting, locating, and correcting faults in a computer program [3]. A large part of the effort of a programmer consists of debugging a program. This makes efficient debugging an important factor for both productivity and program quality [4]. Typically, a defect is detected when a program exhibits unexpected behaviour. In order to locate the cause of such behaviour, it is essential to explain how and why it is generated [5].

A source-level debugger is a very useful and important tool for fault localization that supports the suspension and single-step execution of a program [6]. Single-step execution is based on breakpoints, i.e., points at which execution can be suspended [3]. Stepping through program code allows for a detailed inspection of the program state at a specific point in program execution and the evaluation of the effects of specific code sections.

Debuggers typically are source-level debuggers. However, most debuggers available for agent programs do not provide support for suspending at a particular location in the source code. Instead, these debuggers provide support for suspension at specific points in the reasoning or decision cycle of an agent. The problem is that these points are hard to relate to the agent program code. In addition, these debuggers only show the current state, but do not show the current point in the code where execution will continue. It thus is hard for a programmer to understand how code relates to effects of agent behaviour. Although the role of an agent's decision cycle in the generation of an agent's behaviour is very important, we believe that source-level debugging is also very useful for agent-oriented programming.

In this chapter, we propose a design of a source-level debugger for agent programming. Arguably, such a tool provides an agent programmer with a better understanding of the relation between an agent's program code and its behaviour. Part of the contribution of this chapter is to propose a design approach that is applicable to programming languages for cognitive agents.

2.2. Issues in Debugging Cognitive Agent Programs

In this section, we briefly discuss what is involved in debugging a software system, and analyse the challenges that a developer of cognitive agent programs faces.

2.2.1. Debugging and Program Comprehension

Katz and Anderson [7] provide a model of debugging derived from a general, somewhat simplified model of troubleshooting that consists of four debugging subtasks: (i) program comprehension, (ii) testing, (iii) locating the error, and (iv) repairing the error. Program comprehension, the first subtask in the model, is an important subtask in the debugging process as a programmer needs to figure out why a defect occurs before it can be fixed [8–10]. Gilmore [11] argues that the main aim of program comprehension during debugging is to understand which changes will fix the defect. Based on interviews with developers, Layman *et al.* [12] conclude that the debugging process is a process of iterative *hypothesis refinement* (cf. Yoon and

Garcia [13]). Gathering information to comprehend source code is an important part in the process of hypothesis generation. Lawrance *et al.* [14] also emphasize the information gathering aspect in program comprehension and the importance of *navigating source code*, which they report is by far the most used information source during debugging (cf. Romero *et al.* [6]). Similarly, Eisenstadt [15] suggests to provide a variety of navigation tools at different levels of granularity. In addition, *reproducing the defect* and *inspecting the system state* are essential for fault diagnosis and for identifying the root cause and a potential fix. It is common in debugging to try to replicate the failure [12, 14]. In this process, the expected output of a program needs to be compared with its actual output, for which knowledge of the program's execution and design is required. *Testing* is not only important for reproducing the defect and for identifying relevant parts of code that are involved, but also for verifying that a fix actually corrects the defect [13].

Eisenstadt [15] also argues that it is difficult to locate a fault because a fault and the symptoms of a defect are often far removed from each other (*cause/effect chasm*, cf. Ducassé and Emde [16]). As debugging is difficult, tools are important because they provide insight into the behaviour of a system, enabling a developer to form a mental model of a program [12, 13] and facilitating navigation of a program's source code at runtime [14]. A source-level debugger is a tool that is typically used for controlling execution, setting breakpoints, and manipulating a runtime state. The ability to set breakpoints, i.e., points at which program execution can be suspended [3], by means of a source-level debugger is one of the most useful dynamic debugging tools available and is in particular useful for locating faults [6, 13].

2.2.2. Challenges in Designing a Source-Level Debugger

Even though much of the mainstream work on debugging can be reused, the agent-oriented programming paradigm is based on a set of abstractions and concepts that are different from other paradigms [17, 18]. The agent-oriented paradigm is based on a notion of a *cognitive agent* that maintains a *cognitive state* and derives its choice of action from its *beliefs* and *goals* which are part of this state. Thus, agent-oriented programming is programming with cognitive states.

Compared to other programming paradigms, agent-oriented programming introduces several challenges that complicate the design of a source-level debugger (cf. Lam and Barber [10]). For example, many languages for programming cognitive agents are *rule-based* [19, 20]. In rule-based systems, fault localization is complicated by the fact that errors can appear in seemingly unrelated parts of a rule base [21]. Moreover, a rule base does not define an order of execution. Due to this absence of an execution order, agent debugging has to be based on the specific evaluation strategy that is employed. Moreover, cognitive agent programs repeatedly execute a *decision cycle* which not only *controls the choice of action* of an agent (e.g., which plans are selected or which rules are applied) but also specifies how and when particular *updates of an agent's state* are performed (e.g., how and when percepts and messages are processed, or how and when goals are updated). This style of execution is quite different from other programming paradigms, as a

decision cycle imposes a control flow upon an agent program, and may introduce updates of an agent's state that are executed independently of the program code at fixed places in the cycle or when a state changes due to executing instructions in the agent program. This raises the question of *how to integrate these updates into a single-step debugger*.

An agent's decision cycle provides a set of points that the execution can be suspended at, i.e. *breakpoints*. These points do not necessarily have a corresponding code location in the agent program. For example, receiving a message from another agent is an important state change that is not present in an agent's source, i.e., there is no code in the agent program that makes it check for new messages. Thus, two types of breakpoints can be defined: *code-based* breakpoints and (decision) *cycle-based* breakpoints. Code-based breakpoints have a clear location in an agent program. Cycle-based breakpoints, in contrast, do not always need to have a corresponding code location. Together, these are referred to as the set of *pre-defined* breakpoints that a single-step debugger offers. When single-stepping through a program, these points are traversed. An example[1] of the difference between code-based and cycle-based breakpoints has been illustrated in Figure 2.1. The two traces demonstrate that the same cycle-based event (breakpoint) of achieving the `finished` goal can originate as the result of two different points in the agent program (due to the random execution order of the main module, i.e., either the post-condition of the `finish` action or the `insert` action).

agent's goal
```
1 finished.
```

finish action
```
1 define finish as internal with
2  pre{ not(finished) }
3  post{ finished }
```

agent's main module
```
1 exit = nogoals.
2 order = random.
3
4 module mainModule {
5  if true then finish.
6  if true then insert(finished).
7 }
```

trace example 1
```
1 agent 'example' has been started.
2 'finished' has been adopted as a goal.
3 condition of rule 'if true then finish' holds.
4 pre-condition of 'finish' holds.
5 post-condition 'finished' has been inserted as a belief.
6 'finished' has been achieved and removed as a goal.
7 agent 'example' terminated successfully.
```

trace example 2
```
1 agent 'example' has been started.
2 'finished' has been adopted as a goal.
3 condition of rule 'if true then insert(finished)' holds.
4 'finished' has been inserted as a belief.
5 'finished' has been achieved and removed as a goal.
6 agent 'example' terminated successfully.
```

Figure 2.1: The main components of an exemplary GOAL agent program on the left, and two possible (partial) traces of this agent's execution on the right.

A user should also be able to mark specific locations in an agent's source at which execution will always be suspended, even when not explicitly stepping. To facilitate this, a debugger has to identify such a marker (e.g., a line number) with a code-based breakpoint. These markers are referred to as *user-defined* breakpoints. A

[1] This example uses basic elements from the GOAL language.

specific type of user-defined breakpoint is a *conditional breakpoint*, which only suspends execution when a certain (state) condition applies. A user should also be able to suspend execution upon specific decision cycle events, especially when those do not have a corresponding location in the agent source. This can for example be indicated by a toggle in the debugger's settings. Such an indication is referred to as a *user-selectable* breakpoint.

2.2.3. Languages and Debugging Tools for Cognitive Agents

In this section, we will briefly discuss specific debugging tools as illustrations of state-of-the-art debugging of cognitive agent programs. Moreover, we discuss the main language features and the decision cycle of such an agent program, which is most important in defining the semantics of a language. By understanding the building blocks of a specific agent programming language, we can identify the specific challenges that we will face in designing a source-level debugger for such a language.

We have chosen to focus on some of the more well-known languages in the literature that have been around for some time now and that provide development tools for an agent programmer to code and run an agent system. In our analysis, we have included the rule-based languages 2APL [22], GOAL [23], and Jason [24] and the Java-based languages Agent Factory [25], JACK [26], Jadex [27], and JIAC [28]. The former languages each define their own syntax for rules with conditions expressed in some knowledge representation language such as Prolog, whereas the latter languages build on top of and extend Java with cognitive agent concepts. The rationale for this selection is that we wanted to analyse relatively mature languages that have been well-documented in the literature, whilst making sure we could investigate the current implementation and tools available for a language. It should be noted that published papers about a language may differ from the currently available implementation as most of the languages are being continuously developed.

For the selected platforms, we now describe the *basic language elements and abstractions* available for programming cognitive agents, whether any *embedded languages* are used, e.g., for knowledge representation (KR), and the *decision cycle* that specifies how an agent program is executed. We also summarize the *functionality of the debugging tools* that are available.

2APL

2APL aims for the effective integration of declarative and imperative style programming [22]. To this end, the language integrates *declarative beliefs and goals* with *events* and *plans* that are similar to imperative-style programs in a single rule-based language. JProlog [29] is used as the *embedded KR language*. Plans consist of actions that are composed by a conditional choice operator, iteration operator, sequence operator, or non-interleaving (atomic) operator. If the execution of a plan does not achieve its corresponding declarative goal in the goal base, the goal persists and can be used to select a different plan. A planning goal rule can be used to generate a plan when an agent has certain goals and beliefs. A plan repair

rule can be used when a plan (execution of first action) fails and the agent has a certain belief in its belief base to replace the failed plan with another plan. Events remain in an event base until processed, which includes messages received from other agents. There is no explicit modularization construct available.

Decision Cycle The decision cycle of a 2APL agent is illustrated in Figure 2.2. Each cycle starts by applying all applicable planning goal rules, after which the first action of all plans are executed. Afterwards, all events (first external and then internal) are processed. A new cycle is only started if a rule has been applied or a new event has been received. By using such a cycle, when the execution of a plan fails, it will be either repaired in the same cycle or re-executed in the next.

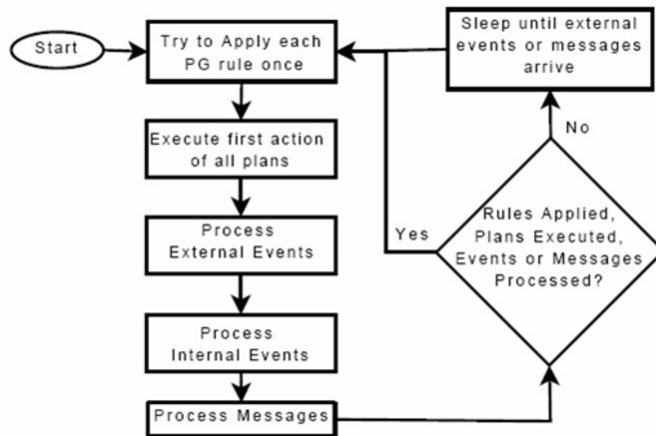


Figure 2.2: The 2APL agent decision cycle [22].

Environment and MAS 2APL agents are connected to an environment via a Java class that implements a dedicated environment interface for the language. The implemented functions form the actions that an agent can execute on the environment's state. Actions can have a return parameter, and thus execution of a plan is blocked until such a return value is available. Actions can also throw exceptions to indicate their failure. Observing the environment is possible by either active or passive sensing. Messages and events are represented through special predicates in an agent's belief base. Each 2APL agent runs in its own single thread, and agents and environments are executed in *parallel*. There are no tools for controlling the scheduling of individual agent execution.

Development Tools 2APL provides a separate runtime with a set of monitoring tools. This runtime environment is separate from the environment for programming a 2APL agent program. During a run, the cognitive state of a single agent can be inspected, though not manipulated, and execution can be controlled by stepping an

entire cycle or suspending execution at specific points in the cycle. While stepping, logging output is generated that can be inspected in a console window. It is also possible to inspect all past states of an agent in this runtime, either in full or for specific bases only. There is no link to the program code that is being executed while stepping or for relating state changes to the execution, although the logging output and state views contain specific code fragments. A user-defined breakpoint mechanism is not available.

Agent Factory

Agent Factory aims to provide a cohesive framework for the development and deployment of multi-agent systems [25]. Agents can be created by implementing a Java interface, but dedicated *rule-based languages* exists as well; through the use of a Common Language Framework (CLF), multiple languages can be used, although the Agent Factory AgentSpeak (AF-AS) language is the default. There are no *embedded languages* used, though it is possible to interface with external APIs written in Java.

Decision Cycle Agent Factory agents follow a specific decision cycle. First, perceptors are fired and beliefs are updated. Second, the agent's desired states are identified, and a subset of desires (new intentions) is added to the agent's commitment set. Older commitments that are of lower importance will be dropped if there are not enough resources available. Finally, various actuators are fired based on the commitments.

Environment and MAS Agent Factory supports the Environment Interface Standard (EIS) [30], a standard for connecting to agent *environments*. Multiple *scheduling algorithms* are available for agents, ranging from round-robin to multi-threaded.

Development Tools Agent Factory provides a separate debugger ("inspection tool") in which the cognitive state of one or more agents can be inspected, though not edited, and execution can be controlled by stepping through entire decision cycles one at a time. It is also possible to inspect all past states of an agent, and a number of logs are provided. However, there is no relation to the code anywhere in this tool. When using a CLF language (instead of plain Java), no user-defined breakpoint mechanism is available.

GOAL

GOAL aims to provide programming constructs for developing cognitive agent programs at the knowledge level that are easy to use, easy to understand, and useful for solving real problems [23]. A dedicated *rule-based language* is used for the formalization of agent concepts. GOAL is designed to allow for any *embedded KR language* to be used; currently mostly SWI-Prolog is used. An agent's cognitive state consists of a belief base, a goal base, a percept base, and a mailbox. Declarative goals specify the state of the environment that an agent wants to establish, and are used to derive an agent's choice of action. Agents commit blindly to their goals, i.e.,

they drop (achieve) goals only when they have been completely achieved. Agents may focus their attention on a subset of their goals by using modules.

Decision Cycle The decision cycle of a GOAL agent is illustrated in Figure 2.3. A GOAL agent cycle starts with the processing of percept rules, allowing an agent to update its cognitive state according to the current perception of the environment. Next, using this new cognitive state, an action is selected for execution. If the precondition of this action holds, its postcondition will be used to update the agent's cognitive state, after which a new decision cycle starts.

Environment and MAS GOAL makes use of EIS [30] to facilitate interaction with *environments*, as does Agent Factory. Agents and environments are executed in *parallel*.

Development Tools A new debugger for GOAL will be designed in this chapter; the previous implementation was similar to that of 2APL, e.g., facilitating the inspection of the cognitive state of a single agent in a separate runtime, and allowing specific steps of the decision cycle to be executed in a stepwise fashion. No relation to the code was provided in this runtime either; multiple consoles with logging output were available. This runtime also has a user-defined breakpoint mechanism, halting the execution when a certain line of code is reached or when a certain condition has been met. However, these breakpoints only paused the agent's decision cycle (i.e., no program code or evaluations were shown). In addition, actions to alter the cognitive state of an agent can be executed, and a cognitive state can be queried as well.

JACK

JACK aims for the elegant and practical development of multi-agent systems [26]. As it is a conservative extension to Java, there is no explicit notion of any *rule-based* constructs. No *embedded language* is used either. Agents are specified by defining the events they handle and send, the data they have, and the plans and capabilities they use. Agents use beliefsets that are relational databases which are stored in memory. Events are used to model messages being received, new goals being adopted, and information being received from the environment. A plan is a recipe for dealing with a given event type, under a certain context condition. Each (Java) statement in a plan body can fail, which will prevent the rest of the plan from being executed, and failure handling will be triggered instead (the consideration of alternative plans). Capabilities and sub-capabilities are used as (hierarchical) modularisation constructs.

Decision Cycle A JACK agent has no explicit decision cycle, but waits until it receives an event or a goal, upon which the agent initiates activity to handle that event or goal; if it does not believe that the event or goal has already been handled, it will look for the appropriate plan(s) to handle it. The agent then executes the plan(s), depending on the event type. Such a plan can succeed or fail; if the plan

fails, the agent may try another plan. The applicability of alternatives is evaluated in the current situation, that is, not the situation when the event was first posted. Moreover, a plan's context condition is split into two parts: the context and a relevance condition, which is used to exclude plans based on the details of an event (which do not change). Meta-plans can also be used to decide which plan to select in more detail (i.e., if multiple are applicable). A special event type is the inference goal, which is handled by executing all applicable plans in sequence.

Environment and MAS JACK has no explicit notion of an *environment*; actions are performed using Java calls. In principle, JACK is *single-threaded*, although specific constructs exist to execute tasks in a new thread.

Development Tools The Jack Development Environment (JDE) allows the creation of entities by dragging and dropping, automatically generating skeleton code. A graphical plan editor is also available, allowing the bodies of plans to be specified using a graphical notation. Moreover, a design tool is included that allows overview diagrams to be drawn, which can be used to create a system's structure by placing entities onto the canvas and linking them together, which can be automatically created based on an existing system as well. A textual trace of processing steps that can be configured to show various types of steps is available as a debugging tool. For distributed agents, interaction diagrams are used that graphically display messages sent between agents. Moreover, graphical plan tracing is provided, showing a graph whilst a plan is executing, highlighting the relevant notes and showing the values of the plan's variables and parameters. Execution can be controlled by stepping through specific events, or stepping with a fixed time delay between the steps. However, a direct relation to the code is absent in all these interfaces, and a user-defined breakpoint mechanism is not available.

Jadex

Jadex aims to make the development of agent based systems as easy as possible without sacrificing the expressive power of the agent paradigm by building up a rational agent layer and allowing for intelligent agent construction using sound software engineering foundations. In its latest version (BDI V3), Jadex uses annotated Java code to designate agent concepts; there are no *rule-based* elements or *embedded languages*. Beliefs are represented in an object-oriented fashion, and operations against a belief base can be issued in a descriptive set-oriented query language. Goals are represented as explicit objects contained in a goal base that is accessible to the reasoning component as well as to plans. An agent can retain goals that are not currently associated to any plan. Four types of goals are supported: perform (actions), achieve (world state), query (internal state), and maintain (continuously ensure a desired state). Thus, changes to beliefs may directly lead to actions such as events being generated or goals being created or dropped. Plans are composed of a head and a body. The head specifies the circumstances under which a plan may be selected, and a context condition can be

stated that must be true for the plan to continue executing. The plan body provides a predefined course of action given in a procedural language. It may access any other application code or third party libraries, as well as the reasoning engine through a BDI API. Capabilities represent a grouping mechanism for the elements of an agent, allowing closely related elements to be put together into a reusable (scoped) module which encapsulate a certain functionality.

Decision Cycle There is also no explicit *decision cycle*, but, similar to JACK, when an agent receives an event, the BDI reasoning engine builds up a list of applicable plans for an event or goal from which candidate(s) are selected and instantiated for execution. Jadex provides settings to influence the event processing individually for event types and instances, though as a default, messages are posted to a single plan, whilst for goals many plans can be executed sequentially until the goal is reached or finally failed (when no more plans are applicable). Selected plans are placed in the ready list, from which a scheduler will execute plans in a step-by-step fashion until it waits explicitly or significantly affects the internal state of the agent (i.e., by creating or dropping a goal). After a plan waits or is interrupted, the state of the agent can be properly updated, e.g., facilitating another plan to be scheduled after a certain goal has been created.

Environment and MAS Jadex offers a standard *environment* model called "EnvSupport" that is meant to support the rapid development of virtual environments. In addition, as Java is used for the procedural code, external APIs can be referenced there as well. A *single thread* model for each component is enforced.

Development Tools Debugging a BDI V3 agent allows stepping an agent through the aforementioned steps that are taken for each event in a separate runtime that does not provide a relation to the program code itself, whilst facilitating inspection of the agent's cognitive state. No modifications to the cognitive state are possible at runtime, and no (generic) logging output or user-defined breakpoint mechanism is available (for BDI V3).

Jason

Jason is a multi-agent system development platform based on an extended version of AgentSpeak [31] aimed at the elegant and practical development of multi-agent systems. A dedicated *rule-based language* is used for the formalization of agent concepts. This language does not make use of any explicit *embedded language*, as KR constructs are part of the agent specification language itself. An agent is defined by a set of beliefs and a set of plans. Thus, a Jason agent is a reactive planning system: (internal or external) events trigger plans. A plan has a head, composed of a trigger event and a conjunction of belief literals representing a context. A plan also has a body, which is a sequence of basic actions or (sub)goals the agent has to achieve (or test) when the plan is triggered. If an action fails or there is no applicable plan for a (sub)goal in the plan being executed, the whole failed plan is removed from the top of the intention, and an internal event associated with that

same intention is generated, allowing a programmer to specify how a particular failure is handled. If no such plan is available, the whole intention is discarded. Two types of goals are distinguished in a goal base: achievement goals and test goals.

Decision Cycle The decision cycle of a Jason agent is illustrated in Figure 2.6. Each cycle, the list of current events is updated, and a single event is selected for processing. For this event, the set of applicable plans is determined, from which a single applicable plan has to be chosen: the intended means for handling the event. Plans for internal events are pushed on top of the current intentions, whilst plans for external events create a new intention. Finally, a single action of an intention has to be selected to be executed in the current decision cycle. When all instructions in the body of a plan have been executed (removed), the whole plan is removed from the intention, and so is the achievement goal that generated it (if applicable). To handle the situation in which there is no applicable plan for a relevant event, a configuration option is provided to either discard such events or insert them back at the end of the event queue. A plan can also be configured for atomic execution, i.e., no other intention may be executed when such a plan has started executing. Moreover, in a cooperative context, the agent can try to retrieve a plan externally.

Environment and MAS Similar to 2APL, a Jason *environment* is a Java class that extends the provided environment interface which contains functions for dealing with percepts and actions. In addition, the Common ARTifact infrastructure for AGents Open environments (CArtAgO) [32] has been developed as a general purpose framework for programming and executing virtual environments. Jason makes use of multiple *threads*. An environment has its own execution thread and uses a configurable pool of threads devoted to executing actions requested by agents. As actions have a return parameter, the execution of a plan is blocked until such a return value is accessible. In addition, each agent has a thread in charge of executing its decision cycle, though these can be configured to be shared in a thread pool as well. Moreover, the agents can use different execution modes. In the default asynchronous mode, an agent performs the next decision cycle as soon as it has finished the current cycle. In the synchronous mode, all agents in a system perform one decision cycle at every "global execution step".

Development Tools Jason provides a separate runtime that includes a debugger. This debugger can show the current and previous cognitive states of an agent, though editing a cognitive state is not possible. It is possible to execute one or more (complete) decision cycles in a stepwise fashion. There is no direct relation to the program code anywhere in this runtime; one general console that displays log messages is available, accompanied by several logging mechanisms that can be used by an agent. Other debugging mechanism such as user-defined breakpoints are not available.

JIAC

JIAC aims to combine agent technology with a service-oriented approach in order to emphasize industrial requirements. A dedicated *rule-based* script-language is used by JIAC: JADL++, although an agent can be programmed in Java as well by using certain pre-defined classes. OWL is used as the *embedded KR language*, i.e., for representing knowledge and beliefs. Agent configurations are provided in XML documents. Services and actions can be described semantically in terms of preconditions and effects, allowing dynamic service discovery and selection. Each agent has a set of abilities (services), which can be used by other agents as well. A specific agent plays a specific role, specified by the relevant goals and actions to fulfil such a role. The actions can be implemented in pure Java, from which other existing technologies like a web service can also be used.

Decision Cycle A JIAC agent uses a *life-cycle*. This life-cycle defines three agents states (void, ready, and active). A specific function can be executed on each state transition. Moreover, a specific "execute method" can be periodically called (in the active state) depending on the agent's configuration.

Environment and MAS JIAC has no explicit notion of an *environment*; actions (and agent communication) are handled through service invocation. Each JIAC agent is run in its own dedicated *thread*, although actions are executed asynchronously.

Development Tools The default Java runtime and/or debugger are to be used for executing JIAC agents. In this case, code written in JADL++ or XML is not (directly) accessible in the debugging process, and no specific agent debugging tools are available, although JIAC does feature several visual tools such as a service designer and a distributed system monitor.

Overview

From this analysis, we can conclude that source-level debugging is not currently employed by any agent programming language. Debugging is performed in a separate runtime application that is able to step through a decision cycle in parts or as a whole. When debugging or running an agent program in 2APL, Agent Factory, GOAL, Jack, Jadex, and Jason, the agent program (i.e., source code) is not shown, and no indication of the currently executed line of code is given. Alternatively, with JIAC agents, debugging is performed at a low level of abstraction (e.g., stepping into code of the framework itself with the Java debugger). 2APL, Agent Factory, and Jason facilitate inspecting an agent's history, for example, stepping 'back' in the agent cycle (i.e., cognitive states), whilst only GOAL supports querying or editing an agent's cognitive state at runtime. GOAL is also the only language that supports some form of user-defined breakpoints in the agent program.

We can thus conclude that current state-of-the-art debuggers for cognitive agent programs provide insight into agent behaviour related to the specific decision cycle it executes, but less so in how the behaviour relates to the agent program code.

However, as this section also showed that relating code to generated behaviour has important benefits for the debugging task, we propose a method for the design of a source-level debugger for cognitive agent programs in the next section.

2.3. Debugger Design Approach

In this section, we propose a design approach for a source-level agent debugging tool that is aimed at providing a better insight into the relationship between program code and the resulting behaviour, with a focus on single-agent debugging. A number of principles and requirements will be introduced to guide the design of a stepping diagram, which defines how the program code is navigated by a user. Such a diagram will be given for both the GOAL and Jason agent programming languages.

2.3.1. Principles and Requirements

We will list some important principles and requirements for a source-level debugger that will be taken into account when designing such a debugger in the next section. As our main objective is to allow an agent developer to detect faults through understanding the behaviour of an agent, an important principle is *usability*. More specifically, Romero *et al.* [6] indicate that a programmer should be able to focus on the declarative semantics of a program, e.g., its rules, checking whether a rule is applicable, how it interacts with other rules, and what role the different parts of a rule play [21, 33]. This is related to the work of Eisenstadt [15], which indicates that a debugger should employ a traversal method for resolving large cause/effect chasms, but without the need to go through indirect steps, intermediate subgoals, or unrelated lines of reasoning. Side-effects pose an additional challenge, as they might be part of a cause/effect chain, but cannot always be easily related to locations in the code. Therefore, *transparency* is an important principle that can be supported by providing a one-to-one mapping between what the user sees and what the interpreter is doing whilst explicitly showing any side effects that occur [34]. A debugger should also strive for temporal, spatial, and semantic *immediacy* [35]. Temporal immediacy means that there should be as little delay as possible between an effect and the observation of related events. Spatial immediacy means that the physical distance (on the screen) between causally related events should be minimal. For example, the evaluation of a rule should be displayed as close as possible to the rule itself. Semantic immediacy means that the conceptual distance between semantically related pieces of information should be kept to a minimum. This is often represented by how many user-interface operations, such as mouse clicks, it takes to get from one piece of information to another. As source-level debuggers aim to correlate code with observed effects, immediacy is an important motivation for the use of such a debugger.

Breakpoints are an essential ingredient of single-step execution. Their main purpose is to facilitate navigating the code and run (generated states) of a program. As discussed in the previous section, a debugger for cognitive agent programming languages can define two types of breakpoints: code-based and cycle-based. We propose that for a source-level debugger, *code-based breakpoints should be pre-*

ferred over cycle-based breakpoints when they serve similar navigational purposes. In other words, when breakpoints show the same state, the code-based breakpoint should be used as a starting point, as it is important to highlight the code to increase a user's understanding of the effects of the program. A good example illustrating this point is the reception of percepts in the decision cycle of a GOAL agent. As percepts are processed in the event module, the entry of this module is a code-based breakpoint that can be identified with the processing of percepts, i.e., the received percepts can be displayed when entering the event module. This reduces the amount of steps that are required and improves the understanding of the purpose of the event module.

In addition, Collier [36] indicates that a user should be able to *control the granularity* of the debugging process. In other words, a user should be able to navigate the code in such a way that a specific fault can be investigated conveniently. For example, a user should be able to skip parts of an agent program that are (seemingly) unrelated to the fault, and examine (seemingly) related parts in more detail. The common way to support this is to define three different step actions: step into, step over, and step out [37]. The stepping flow to follow after each of these actions will have to be defined (i.e., in a stepping (flow) diagram) in order to provide a user with the different levels of granularity that are required.

Hindriks [5] and Romero *et al.* [6] indicate that at any breakpoint, a detailed *inspection of an agent's cognitive state* should be facilitated. The information about an agent's state should be visualized and customizable in multiple ways to support the different kinds of searching techniques that users employ. In addition, the work of Eisenstadt [15] indicates that support for *evaluable cognitive state expressions* should be provided. This will aid a user by supporting, for example, posing queries about specific rule parts to identify which part fails. Romero *et al.* [6] also indicate that modifying the program's state and continuing with a new state should be supported as well. Thus, we propose that support for the *modification of a cognitive state* should be provided. A user could for example be allowed to execute actions in a similar fashion to posing queries in order to perform operations on an agent's state.

2.3.2. Designing a Stepping Diagram

We propose a design approach for a source-level debugger for cognitive agent programs that consists of the following steps. First, possible code-based breakpoints will be defined by using the programming language's syntax (`Step 1`). The relevance of these code-based breakpoints to a user's stepping process needs to be evaluated, leading to a set of points at which events that are important to an agent's behaviour take place (`Step 2a`). In addition, the agent's decision cycle needs to be evaluated for important events that are not represented in the agent's source in order to determine cycle-based breakpoints (`Step 2b`). These points will then be used to define a stepping flow, i.e., identifying the result of a stepping action on each of those points in a stepping diagram (`Step 3`). Finally, other required features such as user-defined breakpoints (`Step 4`), visualization of the execution flow (`Step 5`) and state inspection (`Step 6`) need to be handled. As an example,

we will provide a detailed design for the GOAL agent programming language in this part, and afterwards we will discuss the design of a source-level debugger for some other agent programming languages that were discussed in the previous section as well.

Step 1: Syntax Tree

Inspired by Yoon and Garcia [13], we propose that an agent's syntax tree can be used as the starting point for defining the single-step execution of an agent program. Figure 2.4 (top part) illustrates a slightly modified syntax tree for a GOAL agent, based on the simplified language specification as shown in Tables 2.1 and 2.2 (see Hindriks [38] for the full grammars). Note that '*id*[(*term*)]' represents a call to either a user-specified (environment) action or a module in the grammar (at *action*). In addition, each node in the syntax tree represents a specific type, but not an instance. For example, one module usually consists of multiple rules, as indicated by the labels on the edges. An edge indicates a syntactic link, whilst a broken edge indicates a semantic link. Relevant semantic links need to be added in order to represent program execution flow that is not based on the syntax structure alone.

<table>
<tr><td>module</td><td>:=</td><td>useclause$^+$ option* module id (term) { rule$^+$ }</td></tr>
<tr><td>useclause</td><td>:=</td><td>use id [as usecase] .</td></tr>
<tr><td>usecase</td><td>:=</td><td>knowledge | beliefs | goals | actionspec | module</td></tr>
<tr><td>option</td><td>:=</td><td>exit= exitoption . | focus= focusoption . | order= orderoption .</td></tr>
<tr><td>rule</td><td>:=</td><td>if csq then actioncombo . | forall csq do actioncombo .</td></tr>
<tr><td>csq</td><td>:=</td><td>stateliteral (, stateliteral)*</td></tr>
<tr><td>stateliteral</td><td>:=</td><td>stateop (term) | not (stateop (term)) | true</td></tr>
<tr><td>stateop</td><td>:=</td><td>bel | goal | a-goal | goal-a | percept | sent</td></tr>
<tr><td>actioncombo</td><td>:=</td><td>action (+ action)*</td></tr>
<tr><td>action</td><td>:=</td><td>id[(term)] | generalaction (term)</td></tr>
<tr><td>generalaction</td><td>:=</td><td>insert | delete | adopt | drop | send</td></tr>
<tr><td>term</td><td>:=</td><td>a (composite) KR expression</td></tr>
</table>

Table 2.1: The (simplified) core of the GOAL Module Grammar (BNF).

<table>
<tr><td>specification</td><td>:=</td><td>useclause$^+$ actionspec$^+$</td></tr>
<tr><td>useclause</td><td>:=</td><td>use id [as knowledge] .</td></tr>
<tr><td>actionspec</td><td>:=</td><td>define id[(term)] with pre{ term } post{ term }</td></tr>
<tr><td>term</td><td>:=</td><td>a (composite) KR expression</td></tr>
</table>

Table 2.2: The (simplified) core of the GOAL(User-Defined) Action Spec. Grammar (BNF).

Step 2a: Code-Based Breakpoints

The idea is that each node in a syntax tree can be a possible code-based breakpoint ('step event'). However, as the actual source of some nodes is fully represented

by their children, these non-terminal nodes can be left out of the stepping process. Moreover, some nodes might not be relevant to a user in order to understand an agent's behaviour. Here, we define a node that is *relevant to agent behaviour* as a point at which (i) an agent's cognitive state is inspected or modified, or (ii) a module is called and entered.

State inspections allow a user to identify mismatches between the expected and the actual result of such an inspection. In other words, if a user expects a condition to fail, he should be able to confirm this (and the other way around). Changes to a (cognitive) state are important to the exhibited behaviour of an agent and it should always be possible to inspect it, as should module calls or entries (or similarly, e.g., pushing a plan to an intention) as they are important to the execution flow of an agent. In Figure 2.4, the breakpoints thus identified have been indicated at the corresponding syntax node.

Step 2b: Cycle-Based Breakpoints

There are points at which important behaviour occurs that a user would want to suspend the execution upon that are not present in an agent's syntax tree. For example, achieving a goal involves an important cognitive state inspection (looking for a corresponding belief) and modification (removing the goal), which are not represented in an agent program's source. Points like these that have no fixed correspondence in the agent program we call cycle-based breakpoints. To include such a breakpoint, a toggle (setting) can be added that provides a similar mechanism to user-defined breakpoints by always suspending the execution upon such an event.

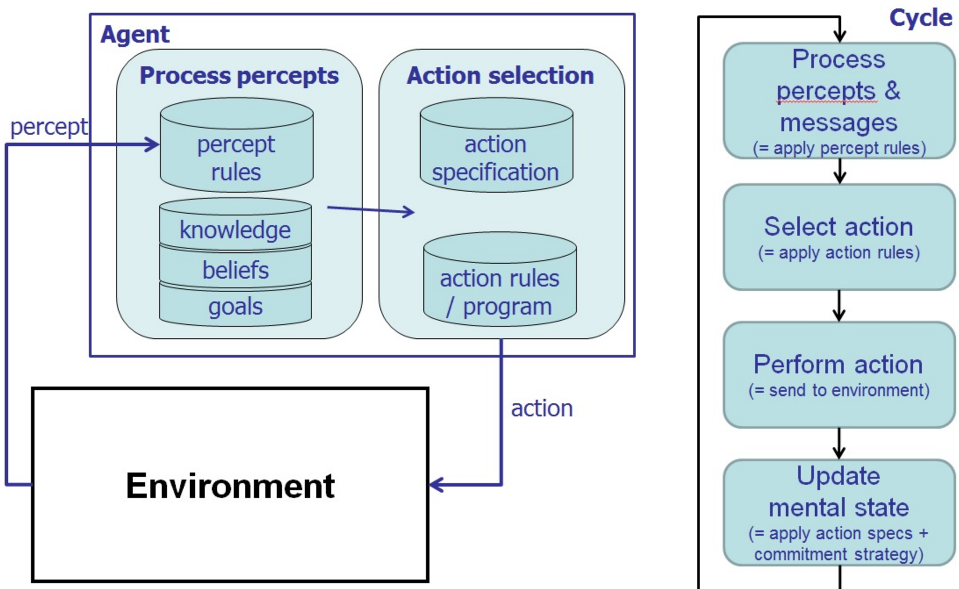


Figure 2.3: The GOAL agent structure and decision cycle [23].

The need for these cycle-based breakpoints and additional explanations highlight an important challenge specific to agent-oriented programming. This results in the fact that we cannot simply construct a source-level debugger by using an agent's source code only. Thus, *a combination of both the syntax and the semantics of an agent is required* to account for all possible changes of an agent's behaviour. The decision cycle of a GOAL agent is illustrated in Figure 2.3. The only event that cannot be directly identified with a location in the source code in GOAL is the achievement of a goal (i.e., in updating the mental state).

Step 3: Stepping Flow

Next, for each identified breakpoint, we need to determine the result of a stepping action, i.e., the flow of stepping. Based on the syntax tree, the stepping actions can be defined as follows:

- **Into:** traverse downward from the current node in the syntax diagram until we hit the next breakpoint. In other words, follow the edges going down in the tree's levels until an indicated node is reached. If the current node is a leaf (i.e., we cannot go down any further), perform an over-step.
- **Over:** traverse to the next node (i.e., to the right) on the current level until we hit the next breakpoint. If there are none, perform an out-step.
- **Out:** traverse upward from the current node until we hit the next breakpoint, whilst remaining in the current context. In other words, the edges going back up in the tree's levels should be traced until any applicable node, and then from there back down again until any indicated node is reached (like an into-step). Here, applicable refers to a 'one-to-many' edge of which not all cases have been processed yet.

On the bottom part of Figure 2.4, the flow for the step into and step over actions on each breakpoint has been illustrated. For readability, the step out action has been left out. Note that the broken edge indicates a link to the event module. This special module is executed after each action that has been performed in order to process any new percepts or messages that have been received by the agent. After the event module has been processed, depending on the rule evaluation order, either the first rule in the module or the rule after the performed action will be evaluated. In addition, a module's exit conditions might have been fulfilled at this point as well, which means that the flow may return to the action combo in which the call to the exited module was made. An example of a stepping flow is illustrated in Figure 2.5.

The extensive definition of an out-step is needed because, for example, when stepping out of a user-defined action, purely following the edges until the previous (upper) breakpoint would result in reaching the module node, whilst we actually want to step to the next rule. Following this reasoning, the same result would be obtained even when doing a step-into from the post-condition node. Therefore, when traversing upward, we consider all nodes. In the example in Figure 2.5, both the action combo and the rule nodes have been processed completely already, so we will reach the module node. If the module contains any more rules, this node will

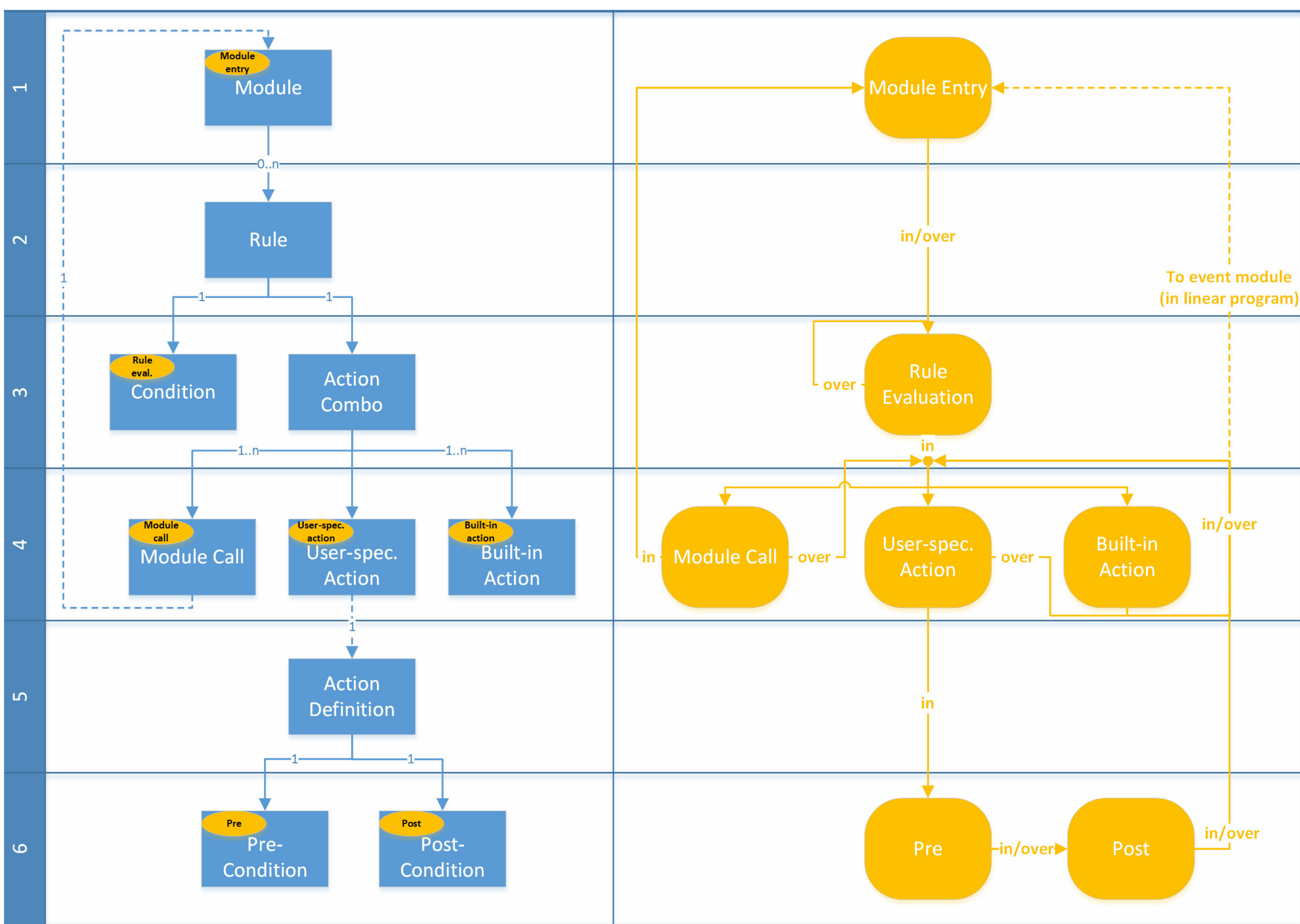


Figure 2.4: A GOAL syntax tree with the relevant breakpoints indicated on the nodes that are present at the different levels on the left side of the figure, and the stepping flow between those breakpoints (for into and over) illustrated on the right side of the figure.

still be applicable, and thus we traverse downward until the first indicated node, which in this case is the next rule evaluation. If there are no more rules to be executed, we continue upwards, exiting the current module entirely, thus arriving back at the point where the call to the module was made (or finishing the execution when in a top-level module).

The stepping flow after a user-selectable breakpoint (i.e., cycle-based) can be dictated by the existing (surrounding) node. For example, achieving a goal is only possible after either executing a cognitive state action or applying a post-condition, so the stepping actions from the relevant node should be used when stepping away from a goal-achieved breakpoint.

Step 4: User-Defined Breakpoints

User-defined breakpoints are usually line-based. In other words, a user can indicate a specific line to break on, instead of a code part. This breaking will always be done, even when not explicitly stepping. Line-based user-defined breakpoints are a widely used mechanism of convenience. However, some breakpoints can be at the same line as other breakpoints. In this case, we pick the breakpoints that are on a higher level in the tree in order to allow a user to still step into a lower level. In the case of GOAL, actions and post-conditions can thus not be used as a ('regular') user-defined breakpoint, whilst module entries, rule evaluations, and pre-conditions can. Conditional user-defined breakpoints in GOAL are also associated with either rule evaluations or pre-conditions (not module entries), but will only suspend execution when the corresponding condition has a successful evaluation (holds).

Step 5: Visualization

Each time the execution is suspended, the code that is about to be executed is highlighted, and any relevant evaluations of (e.g., the values of variables referenced in a rule) of this highlighted code should be displayed. These evaluations will *improve a user's understanding of the execution flow*. For example, if a rule's condition has no solutions, a user will not expect the rule's action to be the next point at which the execution is suspended. Such info is (usually) absent in cycle-based debuggers.

Problems can arise when the code evaluation does not help in making the execution flow clear to a user. For example, stepping into an action's precondition is a step that can lead to a completely different location in the code base, which might be unexpected. Another example is the completion of an action combo, which can result in leaving the current module depending on its exit conditions. To help a user understand these 'jumps' through a program, the code evaluations that are shown can be augmented with additional information indicating the source of the step. For example, when at a precondition, besides the evaluation of the condition a user could also see "selected external action: ...", which gives a hint about the reason why we arrived at the action's precondition. Similar explanations can be provided after or before other steps that might not be clear to a user. Moreover, a visualisation of the call stack (i.e., the locations at which 'functions' were called) is useful for this as well. In the case of GOAL, calls to modules and actions 'push' a new element on the stack, thus keeping the location of the call in the view of the user.

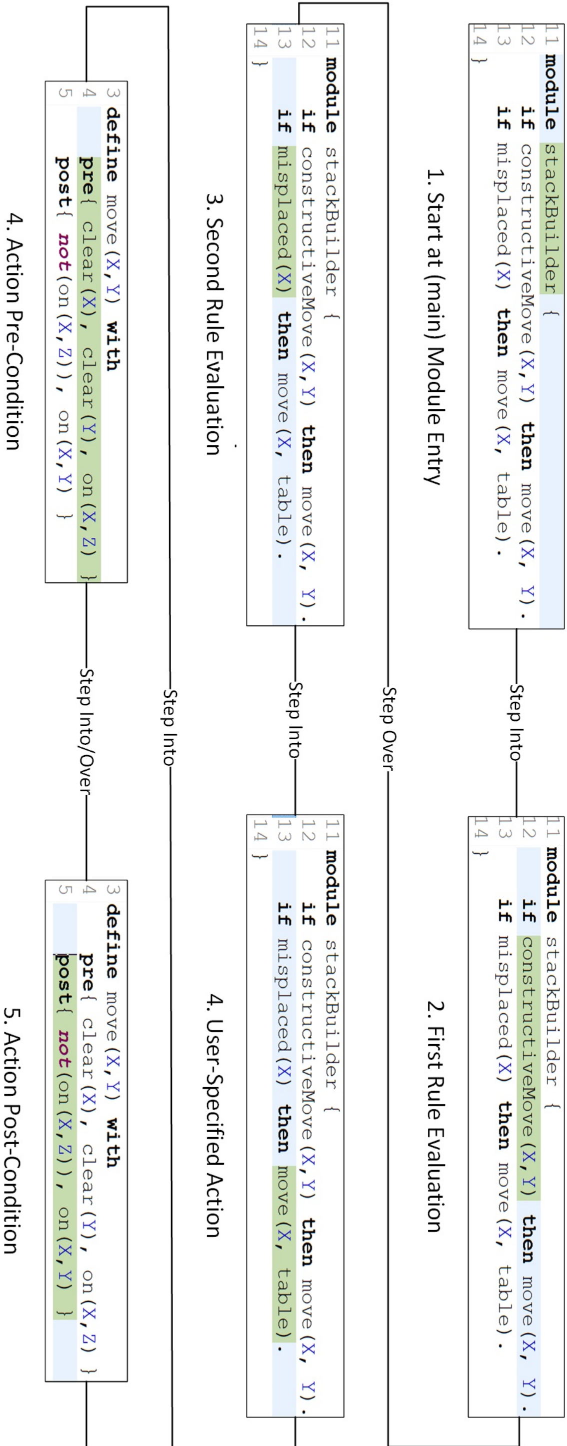


Figure 2.5: An example of a stepping flow through a GOAL agent program.

2

Logging output (i.e., in a console) can also help a user gain understanding of the history of the execution flow. For GOAL, all breakpoints generate a log message that is printed to the specific agent's console. In addition, in case any action fails or an (environment) exception occurs, an error message is shown in the same console.

Step 6: State Inspection

Finally, the inspection and modification of a cognitive state will not be discussed in detail here, as this is a more standard feature. However, care should be taken to conveniently support all of those operations, as they are important to the debugging process. In particular, we have added features that allow the cognitive state of a GOAL agent to be *sorted* and *filtered* (by search queries). This helps a user make sense of a cognitive state, especially if it is very large. In addition, a single interactive console is provided in which both cognitive state queries and actions can be performed in order to respectively *inspect* or *modify* a cognitive state.

2.3.3. Application to Other Agent Programming Languages

The same design steps discussed above can be applied to other agent programming languages in a similar fashion. The syntax and accompanying decision cycle of a Jason agent, for example, can be used in the same manner as described above. Although a Jason agent does not have modules, it does consist of a number of (plan) rules. These rules are built up of a *trigger event*, a *context* (conjunction of belief literals), and a *body* (a sequence of actions: *deeds*). A simplified specification of the syntax of the Jason language is specified in Table 2.3 (see Bordini *et al.* [24] for the full grammar).

agent	::=	*belief** *plan*$^+$
belief	::=	*literal* **.**
plan	::=	*triggering_event* **:** *context* **<-** *body* **.**
triggering_event	::=	(**+** | **-**) [**!** | **?**] *literal*
context	::=	*literal* (**&** *literal*)*
body	::=	*body_formula* (**;** *body_formula*)*
body_formula	::=	[**!** | **?** | **+** | **-**] *literal*
literal	::=	*a (composite) KR expression*

Table 2.3: The (simplified) core of the Jason Agent Specification Grammar (BNF).

The syntax tree (`Step 1`) based on this grammar is illustrated on the top part of Figure 2.7. In that tree, the code-based breakpoints (`Step 2a`) have been indicated as well, i.e., at each node that corresponds with an inspection or modification of an agent's cognitive state. As represented in the decision cycle of a Jason agent in Figure 2.6, Jason has three selection functions (i.e., for events, options, and intentions), a belief revision function, and it also has a goal achievement mechanism and a mechanism for handling events that have no applicable plans; these all represent cycle-based breakpoints (`Step 2b`), as they represent important behaviour that is not present in an agent's syntax tree. A toggle (setting) should be added for each of these events in order to allow suspending execution on them.

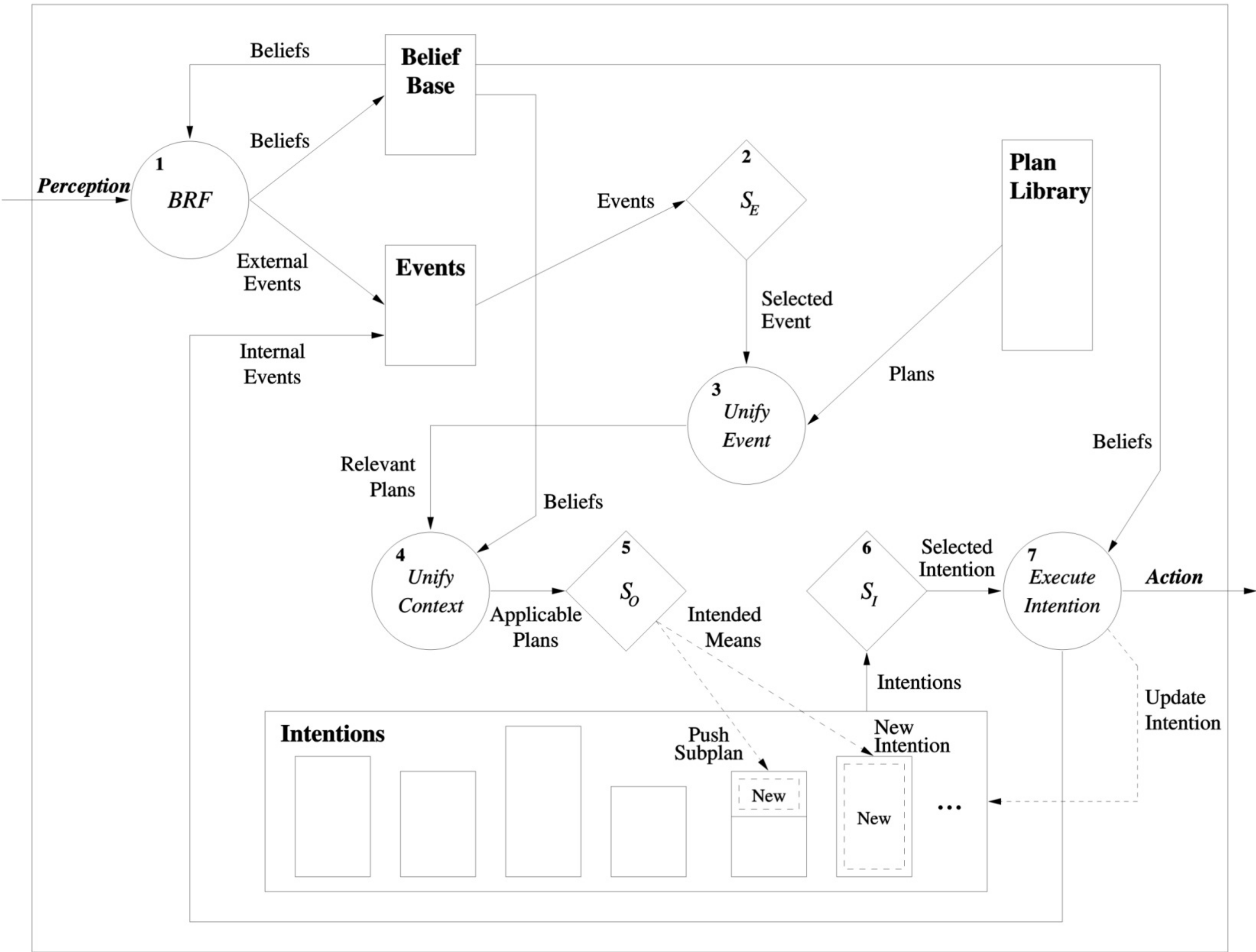


Figure 2.6: The Jason agent decision cycle (based on AgentSpeak) [24].

In addition, we look at a stepping flow for the source-level debugging of Jason agents, as illustrated on the bottom part of Figure 2.7, which has been derived in the same manner as before (`Step 3`). We assume that when for a certain event, the event triggers of an agent's plans are evaluated, a successful evaluation of a trigger will lead to directly evaluating the corresponding context[2]. However, successful evaluation of both an event trigger and the context of a plan will not always directly lead to the execution of the corresponding plan body, as the deeds in the body will be processed into the intention set (by the option selection function), from which in turn a different deed might be selected to execute next (by the intention selection function). The execution of a deed usually leads to a new event, and thus the stepping flow will start again at the first node. Even when a plan is atomic (i.e., indicating that all actions in the plan's body should be executed directly after each other), this flow will remain the same, as atomic plans only override the intention selection mechanism; each deed will still (generally) lead to a new event being generated, and thus the start of a new decision cycle.

In a sense, this flow is similar to that of GOAL. However, after executing an action, the execution flow in GOAL is 'restarted', whilst in Jason an intention that has been selected many cycles ago might still be executed. It will thus be important to make sure the flow from one plan's context into (a certain point in) another plan's body is made as clear as possible, for example by using some visualisation of an agent's intention stack (`Step 5`). For user-defined breakpoints and state inspection (`Step 4` and `Step 6`), the same principles as for GOAL can be applied to Jason.

Finally, for Java-based languages like Jadex, the set of available annotations that indicate the cognitive agent constructs can be used as the base for the syntax tree. In contrast to the default Java debugging flow, the 'evaluation' of such an annotation is an important point of interest. Care would have to be taken to make sure the execution flow between the annotated functions or classes is clear. In general, the design principles and according structure of an agent programming language play a significant role in the design of a source-level debugger for it [**?**]; the harder it is to relate the execution flow of an agent to its program code, the more effort is required to design a debugger that provides sufficient insight into the behaviour resulting from the code.

2.3.4. Implementation for GOAL

An implementation of the proposed source-level debugger design for GOAL was performed by extending the GOAL plug-in for the Eclipse IDE[3]. This plug-in provides a full-fledged development environment for agent programmers, integrating all agent and agent-environment development tools in a single well-established setting [39]. The Eclipse platform is based on an open architecture that allows for building on top of well-known existing frameworks [40]. By using Eclipse and the DLTK framework

[2] Although this does not match the described cycle directly, it is an optimization of the cycle that is the default behaviour of a Jason agent, as confirmed in a discussion with the language designers of Jason. We also assume synchronous execution without the use of concurrent plans.

[3] See `http://goalhub.github.io/eclipse` for a demonstration of the debugger implementation, instructions on how to install GOAL in Eclipse, and links to relevant source code.

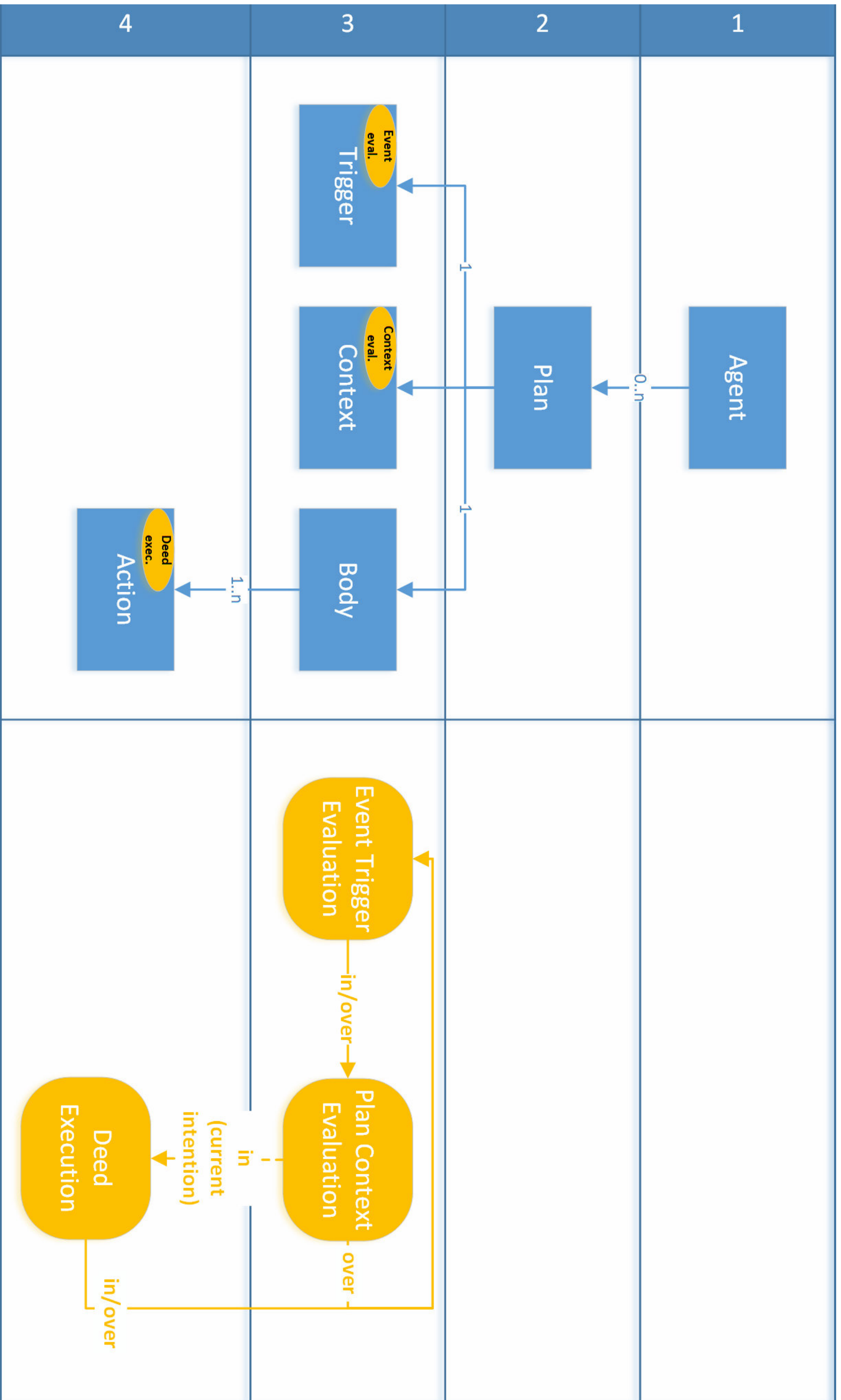


Figure 2.7: A Jason syntax tree with the relevant breakpoints indicated on the nodes that are present at the different levels on the left side of the figure, and the stepping flow between those breakpoints (for into and over) illustrated on the right side of the figure.

[41], for example, a state-of-the-art editor for GOAL has been created, which forms a solid foundation for further tools. It includes a state-of-the-art editor that features syntax highlighting, auto-completion, a code outline, code templates, bracket matching, and code folding. Exchangeable support for embedded KR languages is provided as well.

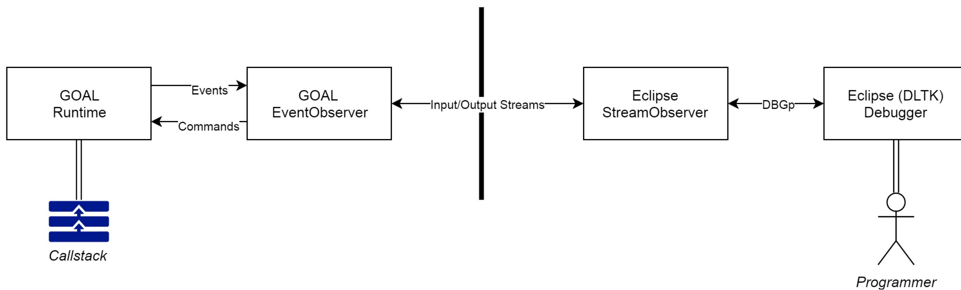


Figure 2.8: An overview of the debugging component structure in GOAL(left) and Eclipse (right).

The source-level debugger implementation makes use of the DeBugGer Protocol (DBGP), which is a common debugger protocol for languages and debugger UI communication [42]. The DLTK framework in Eclipse provides support for using the DBGp protocol in order to offer a debugging interface to a programmer. The debugger implementation expands on the support for implementing the stepping diagram that has been integrated into GOAL by adopting a stack-based execution model, which naturally follows from the different levels that are represented in the stepping tree. This mechanism works by, for example, pushing the task of executing of the actions of a rule (level 5 in the diagram of Figure 2.4) onto the agent's execution stack after successful evaluation of a rule's condition (level 4 in the diagram). The stepping actions (i.e., the flow) can thus be implemented based on these levels, following the rules identified in Step 3 of our approach. This implementation was also inspired by the debugging infrastructure of Lindeman *et al.* [37]. The resulting GOAL debugging interface is illustrated in Figure 2.9. In addition, a general overview of the organization of the debugging components is given in Figure 2.8. As illustrated in this image, a listener is attached to the GOAL runtime. This listener responds to events from that runtime (i.e., the breakpoints), forwarding them to Eclipse (as the GOAL core and Eclipse run in separate processes). In Eclipse, these events are translated into DBGp messages, that in turn control the debugging interface. By using DLTK and DBGp, Eclipse's generic debugging interface could be reused; only slight adjustment was needed to for instance support the inspection of the multiple bases (i.e., beliefs, goals, etc.) of an agent. A user can also give commands through this interface, which traverse the same flow, but then in reverse.

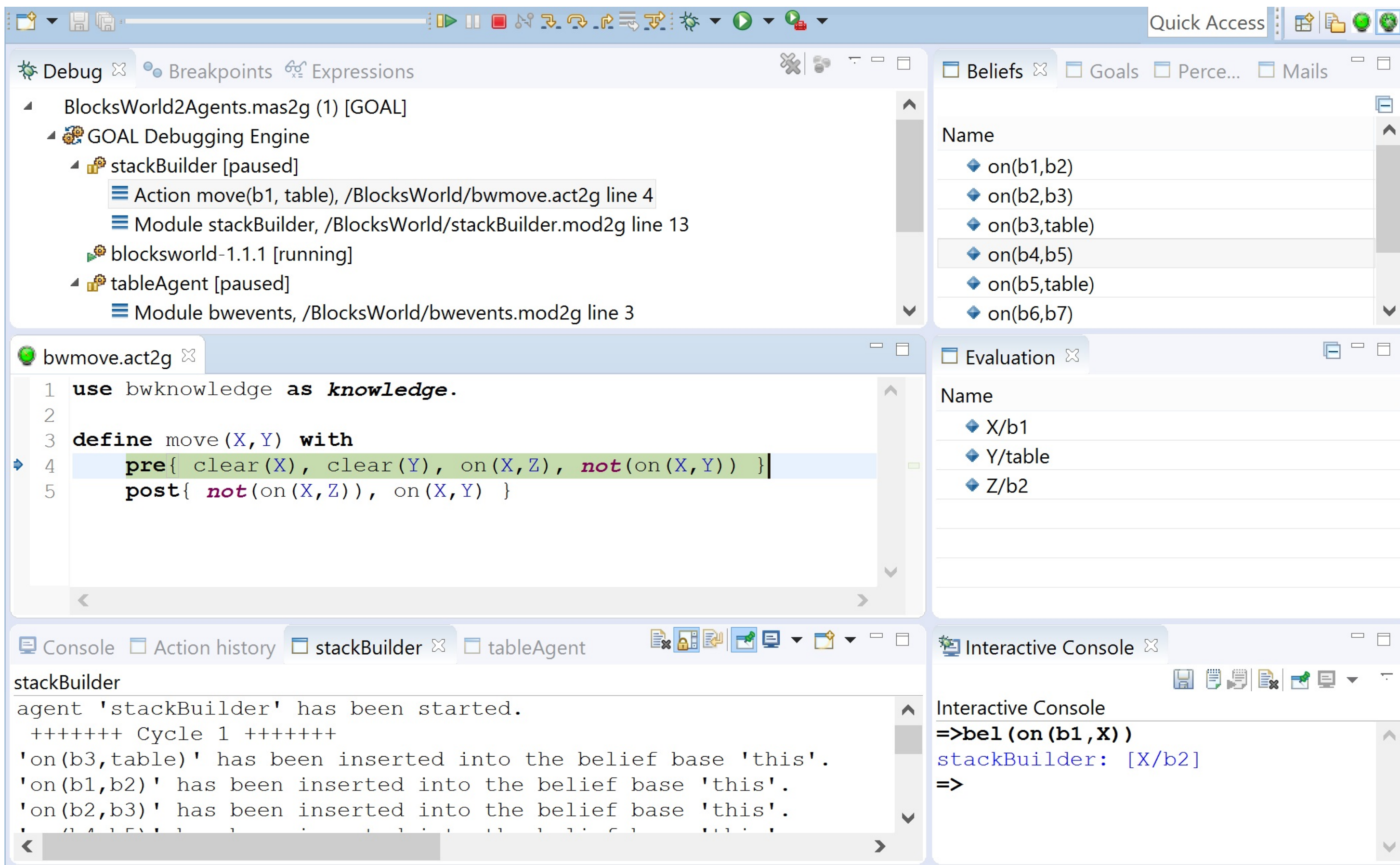


Figure 2.9: A screenshot of the GOAL debugging interface in Eclipse Neon.

2.4. Evaluation

In this section, we evaluate the source-level debugger that has been implemented for GOAL. In order to perform this evaluation, an implementation of the source-level debugger design has been added to the Eclipse plug-in for GOAL.

2.4.1. Quantitative

Table 2.4: Descriptive statistics of the performed evaluation ($N = 94$). The answers to Questions 4 and 8 are discussed in this section.

Question	Mean	Range
1. Total Time Spent	11.1	Number of hours
2. Debugging and Testing	34.5%	Percentage of total time
3. Using Debugger	25.6%	" "
7. Debugger Effectiveness	3.2	1 (not) to 5 (very) effective
5f. Debug: Watch Express.	2.8	1 (least) to 6 (most) useful feature
5b. Debug: Interact. Console	3.0	" "
5d. Debug: Breakpoints	3.0	" "
5a. Debug: Logging	3.7	" "
5e. Debug: State Inspection	4.1	" "
5c. Debug: Stepping	4.3	" "
6f. AOP: Multiple agents	2.2	1 (least) to 6 (most) easy aspect
6b. AOP: Ext. Environments	3.3	" "
6d. AOP: Decision Cycles	3.6	" "
6e. AOP: Rule-based Reas.	3.7	" "
6a. AOP: Use of KR	4.0	" "
6c. AOP: Cognit. States	4.2	" "

A group of over 200 first-year computer science bachelor students at Delft University of Technology made use of this implementation during 6 weeks, working in pairs to develop a team of agents operating in the BW4T environment [43]. When handing in their final agents, the pairs were asked to answer a number of questions in order for us to improve the GOAL platform; it was made clear that their answers would not influence their grade in any way. The questionnaire that was used is provided in Appendix A.

94 pairs filled out this questionnaire. The results were processed by assigning a numeric value to the (Likert-scale) answers on questions 5, 6, and 7, and taking the lower bound as the single number to the answers on questions 1, 2, and 3. Using this processed data, the descriptive statistics given in Table 2.4 were obtained. Note that for readability the order in this table does not correspond to the order in which the questions were given (e.g., the ordering questions 5 and 6 have been sorted in ascending order of their results). In addition, (Tukey) boxplots for these results are given in Figures 2.10, 2.11, and 2.12.

The results show that these novice agent programmers spend about a third of their time on debugging and testing their agent program, and nearly all that time

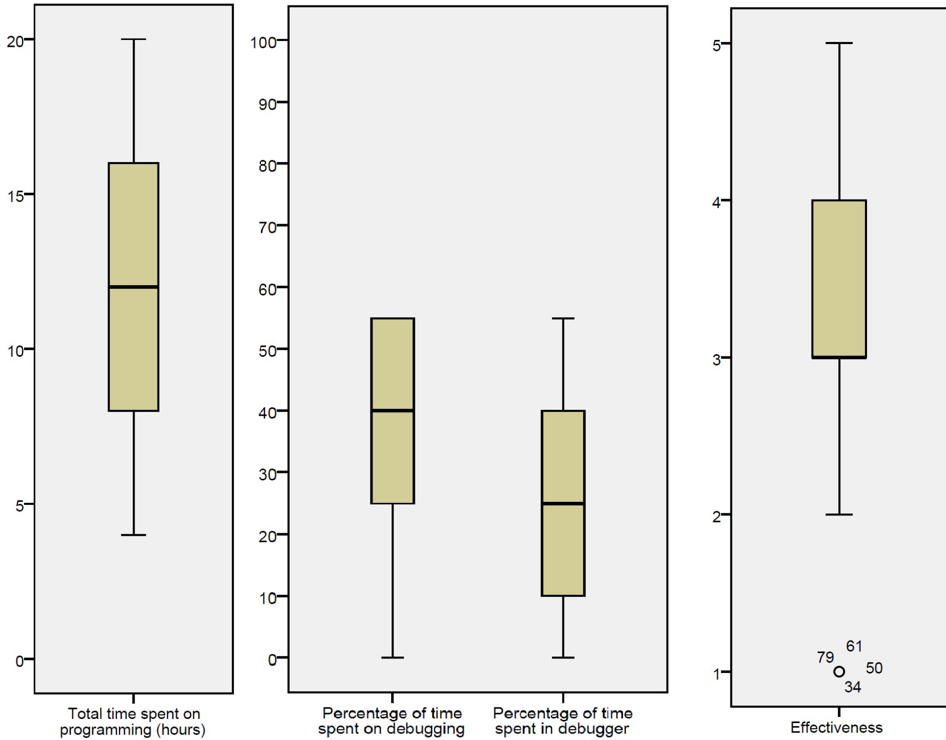


Figure 2.10: Boxplots of the results of Questions 1, 2, 3, and 7 (higher means more effective).

(a quarter of the total time on average) is spent using the source-level debugger. Stepping and state inspection are clearly regarded as the most useful features of the debugger, which is a positive affirmation of our work. The debugger is usually utilized (*Q4*) when students see their agent doing something wrong (56% of respondents indicated this) or to see how their agent program behaves exactly (32% of respondents indicated this). In addition, the use of external environments and especially multiple agents is regarded as causing most of the problems for debugging (*Q5*), suggesting directions for future work.

A correlation analysis showed some significant results, i.e., with a Pearson correlation coefficient less than .05 in a 2-tailed test, as shown in more detail in Appendix A. An expected result is that the the total time spent on programming (*Q1*) is positively correlated with the percentage of time spend on debugging and testing (*Q2*), which in turn is positively correlated with the percentage of time spend in the debugger (*Q3*). Moreover, the percentage of time spend in the debugger is positively correlated with the perceived effectiveness of the debugger (*Q7*). As the total average of the perceived effectiveness of source-level debugging for locating faults is quite high (3.2 out of 5 even with some low outliers), this correlation suggests that *although some time is needed to familiarize oneself with the agent debugging*

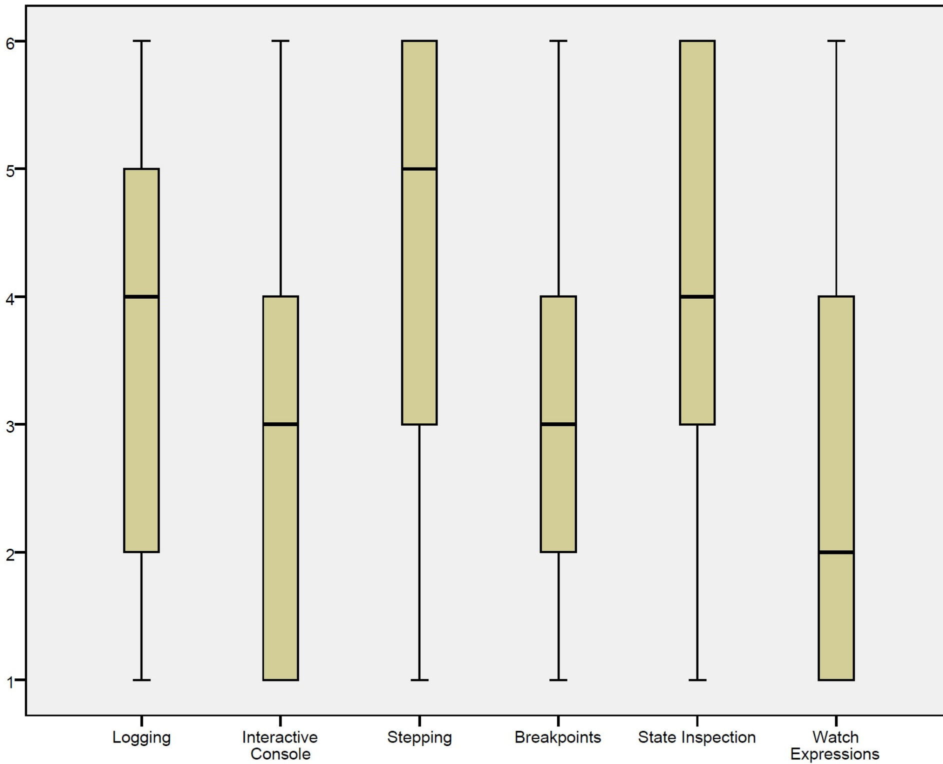


Figure 2.11: Boxplot of the results of Question 5 (higher means a more useful debugging feature).

tools, they provide a programmer with an effective development tool.

In addition, the correlations between the different debugging features (*Q5*) suggest that *the students can be split into two groups*: one that indicates stepping, state inspection and breakpoints are most useful, and one that indicates logging, the interactive console, and watch expressions are most useful. This grouping is supported by the correlations between the debugging features and the different AOP aspects (*Q6*), as finding stepping and/or state inspection a more useful debugging feature is positively correlated with the rule-based aspect of AOP (i.e., regarded as easier), whilst in contrast finding logging and/or the interactive console a more useful debugging feature is negatively correlated with the rule-based aspect of AOP (i.e., regarded as more difficult). This suggests that *using source-level debugging facilitates understanding of rule-based reasoning*. A related result is the fact that finding breakpoints a more useful debugging feature is negatively correlated with the usefulness of the state inspection debugging feature, and that finding watch expressions a more useful debugging feature is negatively correlated with the logging debugging feature. This suggests that *breakpoints and watch expressions are useful tools to reduce the amount of 'manual' inspection* (e.g., looking at states or logs) that is needed.

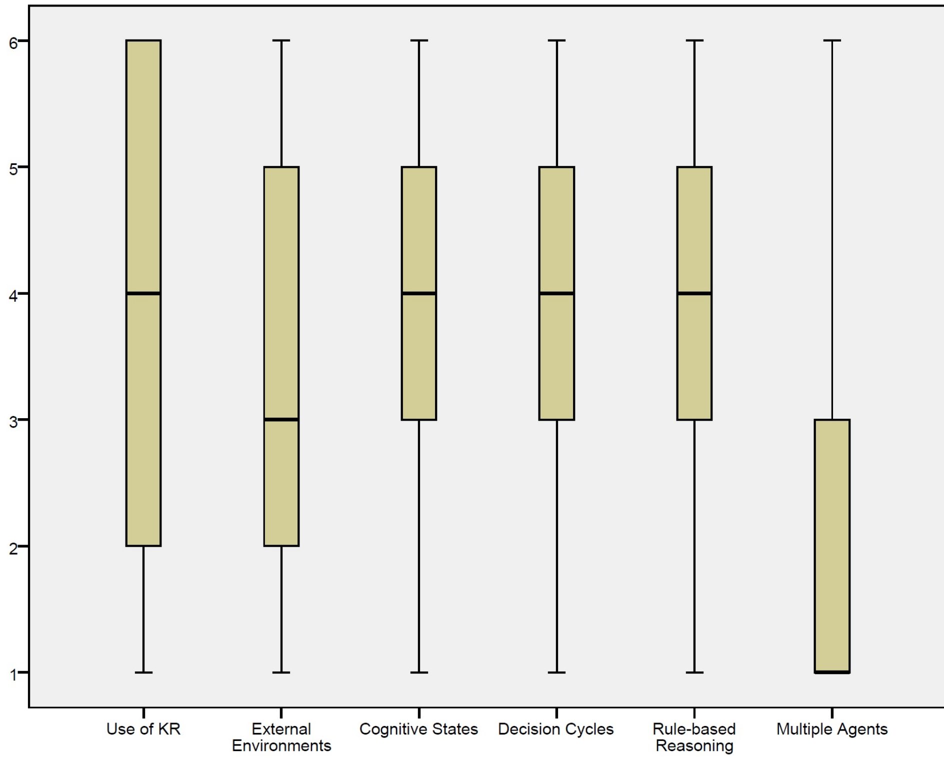


Figure 2.12: Boxplot of the results of Question 6 (higher means a more easy AOP aspect).

2.4.2. Qualitative

The hypotheses in this section are also supported by the qualitative data that was provided by the students in their answers to Question 8. A selection of these comments is given below:

- *"I liked the visual GOAL environment as it is much more relatable than a simple console. I liked its simplicity in separating logical processes."*
- *"Debugging with multiple agents was not as good as we expected it to. It was hard to see what each agent believed and percepted. You had to pause one agent, but the others would just continue to gather blocks. So by trying to debug you interfered with the program and you might get a different outcome that way. That is not what you want to happen when you are debugging."*
- *"Being able to look at the beliefs, goals, percepts and messages when the bots are paused was very helpful for debugging, as was seeing those update while stepping through a bot's code. This especially helped with starting to learn the platform, as it was easy to see the effect of each line of code."*
- *"I found the debugger ineffective as when you use the debugger the agents*

act differently from without using the debugger. Also because when one agent hits a breakpoint the others keep going, making it very hard to determine what went wrong in the communication or teamwork."

- *"We loved the debugging, this was really easy to understand and you could easily find the bugs. The pausing and stepping is clear and very useful!"*
- *"I found it very annoying that the debugger and stepping apparently could yield very different behaviour, but it was something that you could work around with eventually. In a way it was a good thing because it also reminds you of its multi-threaded nature; make no assumptions about synchronization. I think this is partly what caused the most trouble for people using GOAL. Using the debugger is an essential tool for figuring out when a rule is fired, namely by putting a (conditional) breakpoint at the 'then'-line. I liked this a lot!"*

Although the comments are generally positive, clear directions for future work (as also suggested by the quantitative data) are indicated by the students, which will be discussed in the next section.

2.5. Conclusions and Future Work

In this chapter, we proposed a source-level debugger design for agents that takes code stepping more serious than existing solutions, aimed at providing a better insight into the relationship between program code and the resulting behaviour. We identified two different types of breakpoints for agent programming: code-based and cycle-based. The former are based on the structure of an agent program, whereas the latter are based on an agent's decision cycle. We proposed concrete design steps for designing a debugger for cognitive agent programs. By using the syntax and decision cycle of an agent programming language, a set of predefined breakpoints and a flow between them can be determined in a structured manner, and represented in a stepping diagram. Based on such a diagram, features such as user-defined breakpoints, visualization of the execution flow, and state inspection can be handled. We provided a concrete design for the GOAL and Jason programming languages, as well as a full implementation for GOAL, and argue that our design approach can be applied to other agent programming languages as well. A qualitative evaluation shows that agent programmers prefer the source-level (i.e., code-based) over a purely cycle-based debugger.

The debugging challenges related to rule-based reasoning and agent decision cycles form the core of this chapter. However, there are more challenges in debugging cognitive agents that need to be addressed. One of these is the fact that agents are (usually) connected to an *environment*. Two problems need to be dealt with: (i) it cannot be assumed that an environment is *deterministic* which makes it difficult to reproduce a defect, and (ii) environments typically cannot be *suspended instantly* (or at all) which makes it difficult to understand the context of a defect. This is especially the case when dealing with physical environments, e.g., controlling

robots like search-and-rescue drones. Simulating environments could be a possible solution for this, i.e., using a deterministic, suspendable and repeatable version of an environment for debugging purposes. However, this is a major challenge, especially in large or uncertain domains.

Another problem is the fact that debugging *multiple agents* at once is significantly more complicated than debugging a single agent. This problem is most prominent in the evaluation results. Although debugging concurrent programs is a major problem in any type of programming language [44], the agent-oriented paradigm entails a number of aspects that might aid in supporting this for multi-agent systems specifically. For example, the fact that the way in which agents communicate is determined by the platform could be exploited for specific visualizations. In addition, grouping concepts such as organizations and roles [45, 46] could help in clustering information for users, especially considering that the amount of information needed for debugging can easily explode in a systems with many agents.

In addition, many programming languages for cognitive agents embed *knowledge representation (KR) languages* like Prolog or a Web Ontology Language (OWL). Some agent programming languages also embed (instead of extend) an object-oriented programming language such as Java. This introduces the additional problem of how to employ the debugging frameworks that are available for the embedded languages. For example, the SWI Prolog trace mechanism could be made available through the GOAL debugger in some way.

Finally, the debugger design should be evaluated on different groups of users, i.e., different from novice (first-year student) programmers. Moreover, additional quantitative measures such as the average time of finding a bug, the average quality of programs (e.g., the amount of faults/failures in a result with or without use of the source-level debugger), and more could give more insights into the challenges of debugging multi-agent systems.

References

[1] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, *Designing a source-level debugger for cognitive agent programs,* Autonomous Agents and Multi-Agent Systems **31**, 941 (2017).

[2] V. J. Koeman and K. V. Hindriks, *Designing a source-level debugger for cognitive agent programs,* in *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, edited by Q. Chen, P. Torroni, S. Villata, J. Hsu, and A. Omicini (Springer International Publishing, Cham, 2015) pp. 335–350.

[3] ISO/IEC/IEEE, *24765:2017-9 Systems and software engineering – Vocabulary,* https://www.iso.org/standard/71952.html (2017).

[4] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009).

[5] K. V. Hindriks, *Debugging is explaining,* in *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 7455, edited by I. Rahwan, W. Wobcke, S. Sen, and T. Sugawara (Springer Berlin Heidelberg, 2012) pp. 31–45.

[6] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant, *Debugging strategies and tactics in a multi-representation software environment,* International Journal of Human-Computer Studies **65**, 992 (2007).

[7] I. R. Katz and J. R. Anderson, *Debugging: An analysis of bug-location strategies,* Human–Computer Interaction **3**, 351 (1987).

[8] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, *A systematic survey of program comprehension through dynamic analysis,* IEEE Transactions on Software Engineering **35**, 684 (2009).

[9] D. N. Lam and K. S. Barber, *Comprehending agent software,* in *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05 (ACM, New York, NY, USA, 2005) pp. 586–593.

[10] D. N. Lam and K. S. Barber, *Debugging agent behavior in an implemented agent system,* in *Programming Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 3346, edited by R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni (Springer Berlin Heidelberg, 2005) pp. 104–125.

[11] D. J. Gilmore, *Models of debugging,* Acta Psychologica **78**, 151 (1991).

[12] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, *Debugging revisited: Toward understanding the debugging needs of contemporary software developers,* in *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on* (2013) pp. 383–0392.

[13] B.-d. Yoon and O. Garcia, *Cognitive activities and support in debugging,* in *Human Interaction with Complex Systems. Proceedings., Fourth Annual Symposium on* (1998) pp. 160–169.

[14] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. Fleming, *How programmers debug, revisited: An information foraging theory perspective,* Software Engineering, IEEE Transactions on **39**, 197 (2013).

[15] M. Eisenstadt, *My hairiest bug war stories,* Communications of the ACM **40**, 30 (1997).

[16] M. Ducassé and A.-M. Emde, *A review of automated debugging systems: Knowledge, strategies and techniques,* in *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88 (IEEE Computer Society Press, Los Alamitos, CA, USA, 1988) pp. 162–171.

2

[17] K. V. Hindriks, *The shaping of the agent-oriented mindset,* in *Engineering Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 8758, edited by F. Dalpiaz, J. Dix, and M. B. van Riemsdijk (Springer International Publishing, 2014) pp. 1–14.

[18] J. Sudeikat, L. Braubach, A. Pokahr, W. Lamersdorf, and W. Renz, *Validation of BDI agents,* in *Programming Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 4411, edited by R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni (Springer Berlin Heidelberg, 2007) pp. 185–200.

[19] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, eds., *Multi-Agent Programming: Languages, Platforms and Applications* (Springer US, 2005).

[20] A. E. F. Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, eds., *Multi-Agent Programming: Languages, Tools and Applications* (Springer US, 2009).

[21] V. Zacharias, *Tackling the debugging challenge of rule based systems,* in *Enterprise Information Systems*, Lecture Notes in Business Information Processing, Vol. 19, edited by J. Filipe and J. Cordeiro (Springer Berlin Heidelberg, 2009) pp. 144–154.

[22] M. Dastani, *2APL: a practical agent programming language,* Autonomous Agents and Multi-Agent Systems **16**, 214 (2008).

[23] K. V. Hindriks, *Programming rational agents in GOAL,* in *Multi-Agent Programming: Languages, Tools and Applications*, edited by A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini (Springer US, 2009) pp. 119–157.

[24] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak using Jason* (John Wiley & Sons, Ltd, 2007).

[25] C. Muldoon, G. M. O'Hare, R. W. Collier, and M. J. O'Grady, *Towards pervasive intelligence: Reflections on the evolution of the Agent Factory framework,* in *Multi-Agent Programming: Languages, Tools and Applications*, edited by A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini (Springer US, 2009) pp. 187–212.

[26] M. Winikoff, *Jack intelligent agents: An industrial strength platform,* in *Multi-Agent Programming*, Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15, edited by R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni (Springer US, 2005) pp. 175–193.

[27] A. Pokahr, L. Braubach, and W. Lamersdorf, *Jadex: A BDI reasoning engine,* in *Multi-Agent Programming*, Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15, edited by R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni (Springer US, 2005) pp. 149–174.

[28] B. Hirsch, T. Konnerth, and A. Heßler, *Merging agents and services - the JIAC agent platform,* in *Multi-Agent Programming: Languages, Tools and Applications*, edited by A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini (Springer US, 2009) pp. 159–185.

[29] B. Demoen and P. Tarau, *jProlog, a Prolog to Java compiler,* https://people.cs.kuleuven.be/~bart.demoen/PrologInJava (1996).

[30] T. M. Behrens, K. V. Hindriks, and J. Dix, *Towards an environment interface standard for agent platforms,* Annals of Mathematics and Artificial Intelligence **61**, 261 (2011).

[31] A. S. Rao, *AgentSpeak(L): BDI agents speak out in a logical computable language,* in *Agents Breaking Away*, Lecture Notes in Computer Science, Vol. 1038, edited by W. Van de Velde and J. W. Perram (Springer Berlin Heidelberg, 1996) pp. 42–55.

[32] A. Ricci, M. Piunti, M. Viroli, and A. Omicini, *Environment programming in CArtAgO,* in *Multi-Agent Programming: Languages, Tools and Applications*, edited by A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini (Springer US, 2009) pp. 259–288.

[33] V. Zacharias and A. Abecker, *Explorative debugging for rapid rule base development,* in *Proceedings of the 3rd Workshop on Scripting for the Semantic Web at the ESWC* (2007).

[34] T. Rajan, *Principles for the design of dynamic tracing environments for novice programmers,* Instructional Science **19**, 377 (1990).

[35] D. Ungar, H. Lieberman, and C. Fry, *Debugging and the experience of immediacy,* Communications of the ACM **40**, 38 (1997).

[36] R. Collier, *Debugging agents in Agent Factory,* in *Programming Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 4411, edited by R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni (Springer Berlin Heidelberg, 2007) pp. 229–248.

[37] R. T. Lindeman, L. C. Kats, and E. Visser, *Declaratively defining domain-specific language debuggers,* SIGPLAN Notices **47**, 127 (2011).

[38] K. V. Hindriks, *Programming cognitive agents in GOAL,* https://bintray.com/artifact/download/goalhub/GOAL/GOALProgrammingGuide.pdf (2018).

[39] V. J. Koeman and K. V. Hindriks, *A fully integrated development environment for agent-oriented programming,* in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection*, Lecture Notes in Computer Science, Vol. 9086, edited by Y. Demazeau, K. S. Decker, J. Bajo Pérez, and F. de la Prieta (Springer International Publishing, 2015) pp. 288–291.

[40] D. Geer, *Eclipse becomes the dominant Java IDE,* Computer **38**, 16 (2005).

[41] S. Gomanyuk, *An approach to creating development environments for a wide class of programming languages,* Programming and Computer Software **34**, 225 (2008).

[42] S. Caraveo and D. Rethans, *DBGP - a common debugger protocol for languages and debugger UI communication,* https://xdebug.org/docs-dbgp.php (2007).

[43] M. Johnson, C. Jonker, B. Riemsdijk, P. J. Feltovich, and J. M. Bradshaw, *Engineering societies in the agents world x: 10th international workshop, esaw 2009, utrecht, the netherlands, november 18-20, 2009. proceedings,* (Springer Berlin Heidelberg, 2009) Chap. Joint Activity Testbed: Blocks World for Teams (BW4T), pp. 254–256.

[44] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal, *10 years of research on debugging concurrent and multicore software: A systematic mapping study,* Software Quality Journal **25**, 49 (2017).

[45] H. Aldewereld and V. Dignum, *OperettA: Organization-oriented development environment,* in *Languages, Methodologies, and Development Tools for Multi-Agent Systems: Third International Workshop, LADS 2010, Lyon, France, August 30 – September 1, 2010, Revised Selected Papers*, edited by M. Dastani, A. El Fallah Seghrouchni, J. Hübner, and J. Leite (Springer Berlin Heidelberg, 2011) pp. 1–18.

[46] M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat, *MOISE: An organizational model for multi-agent systems,* in *Advances in Artificial Intelligence: International Joint Conference 7th Ibero-American Conference on AI 15th Brazilian Symposium on AI IBERAMIA-SBIA 2000 Atibaia, SP, Brazil, November 19–22, 2000 Proceedings*, edited by M. C. Monard and J. S. Sichman (Springer Berlin Heidelberg, 2000) pp. 156–165.

3

Automating Failure Detection in Cognitive Agents

Debugging is notoriously difficult and time consuming but also essential for ensuring the reliability and quality of a software system. In order to reduce debugging effort and enable automated failure detection, we propose an automated testing framework for detecting failures in cognitive agent programs. Our approach is based on the assumption that modules within such programs are a natural unit for testing.

We identify a minimal set of temporal operators that enable the specification of test conditions and show that the test language is sufficiently expressive for detecting all failure types of an existing failure taxonomy. We also introduce an approach for specifying test templates that supports a programmer in writing tests.

Furthermore, empirical analysis of agent programs allows us to evaluate whether our approach using test templates adequately detects failures, and to determine the effort that is required to do so in both single and multi agent systems. We also discuss a concrete implementation of the proposed framework for the GOAL *agent programming language that has been developed for the Eclipse IDE. With the use of this framework, evaluations have been performed based on test files and according questionnaires that were handed in by 94 novice programmers.*

This chapter has been published in the International Journal of Agent-Oriented Software Engineering **6**(3-4) (2018) [1], an extension of work published in the book Engineering Multi-Agent Systems (2016) [2], in turn an extension of works published in the Conference on Autonomous Agents and Multi-Agent Systems (2016) [3, 4].

3.1. Introduction

Debugging is notoriously difficult and extremely time consuming [5] but also essential for ensuring the reliability and quality of a software system. Manual testing, using, for example, a debugger for single-step execution to identify differences between observed and intended behaviour, however, is not an efficient failure detection method and heavily relies on the programmer to identify the failure. In order to reduce debugging effort and enable automated failure detection, we propose an *automated testing framework for cognitive agent programs*. Automated testing yields a reduction in the effort needed to detect a failure and is more effective than code inspection methods [6]. In addition, it also facilitates running tests repeatedly at no additional costs.

The aim of this chapter is to introduce and develop a testing framework that supports *automated failure detection* for programs written in rule-based agent programming languages that use logic for representing the cognitive state of the agent, including e.g. its beliefs. A *failure* is an event in which a system does not perform a required function within specified limits [7]. Failures thus are manifestations of undesired behaviour. They are caused by a *fault*, an incorrect step, process, or data definition in a program [7] or mistake in a program [8]. Upon detecting a failure, a programmer needs to locate and correct the fault that causes the failure. Our focus is on automating the detection of failures and on dynamic analysis, i.e., the process of evaluating a system or component based on its behaviour during execution [7]. The testing framework that we introduce, however, also provides support for fault localization.

The main contribution of this chapter is an *automated testing framework* for cognitive agent programs that provides support for detecting frequently occurring failure types. As a first step towards such a framework, we argue that an aggregate level that collects multiple goals, plans, and/or rules in a single unit is the most natural unit for testing, and that test conditions should be associated with such units, which we call *modules*. Second, we introduce two basic temporal operators that in practice turn out to be sufficient for specifying test conditions to detect failures. Third, using this generic framework, we propose test templates for failure types that have been identified in a previously developed taxonomy by Winikoff [8]. The test templates can be considered as a refinement of this failure taxonomy. Finally, we introduce a test approach for deriving test templates given some initial functional requirements.

In order to empirically evaluate and demonstrate that our framework is expressive enough to detect all failure types, we verify that we can reproduce and identify all failures found in the sample used by Winikoff [8]. Moreover, we show that by automating testing, we are able to identify more failures. We also show that our work is not biased towards this sample by demonstrating that the same test approach is able to identify failures in different samples of programs. In addition, we discuss empirical and qualitative feedback that was collected, focusing on the practical use by novice programmers of a concrete implementation of the proposed framework.

As cognitive agents usually operate in dynamic, asynchronous environments

[9], they need to be able to achieve their objectives flexibly and robustly. Winikoff [10] finds that because of this, agent programs are harder to test than equivalently sized procedural programs. Therefore, we also evaluate the effort that is required to reproduce failures in both single and multi agent systems. Winikoff [10] also states that it is not yet possible to apply formal validation methods to realistically-sized agent systems, providing further motivation for this work, focusing on practical run-time validation.

3.2. Related Work

In general, different techniques for detecting failures of program code are available, ranging from *inspection* of source code and logs to automated *testing* tools [6]. The need for debugging techniques and test approaches for agent-oriented programming has been broadly recognized [11–13]. Techniques for agent-oriented programming need to be based on the underlying agent paradigm [14, 15]. However, this is a significant challenge, as they should for example take into account that agents execute a specific decision cycle and operate in non-deterministic environments [16–18].

To facilitate code inspection, a (source-level) debugger that supports single-step execution of an agent can be used (c.f. Chapter 2). Debugging is particularly useful for zooming in on a bug of which the location is already more or less clear; it requires a programmer to go through the execution steps of an agent program one-by-one and to observe any mistakes in the program manually. This method of code inspection is not only inefficient, but also subjective, as it depends on observations made by a programmer. It requires the same effort repeatedly since, after correcting an identified fault, a programmer needs to manually evaluate program behaviour again to verify that a fix does not introduce new defects. Automated testing offers a method that is complementary to debugging, able to find different kinds of failures, and thus increasing the overall number of detected failures [5, 6]. Moreover, debugging techniques that support the localization of the fault that causes a failure that has been identified by automated tests are still needed. Such techniques include state inspection, mechanisms for browsing and searching (historical) agent states, et cetera [19] (c.f. Chapter 2).

Five levels of testing a multi-agent system (MAS) are distinguished by Moreno *et al.* [20]: *unit*, *agent*, *integration*, *system*, and *acceptance*. In this classification, unit testing targets components that are part of an agent. Testing an agent means to test the integration of these components as well as its ability to interact with its environment. Integration testing focuses on agent interaction and communication protocols, system testing concerns the target operating environment, and acceptance testing takes the customer's perspective into account. The focus of this chapter is on unit and agent testing.

Testing can target different artifacts [6]. The testing framework of Zhang *et al.* [15], for example, targets Prometheus design models. The paper presents a mechanism for generating suitable test cases. Similarly, the methodology of Gómez-Sanz *et al.* [21] is based on specific 'meta-models' and 'protocol descriptions'. Other methods focus on the interaction between agents specifically, often using 'mock

agents' to check if a specific interaction has taken place [22, 23]. The concept of a 'mock agent', i.e., a specific agent that is used for testing, is also used in the work of Carrera *et al.* [24] to test an agent's behaviour based on 'agent stories'. Most of these approaches consider an agent to be the smallest possible artifact to test. Other approaches target more specific artifacts like plans [25, 26], goals [27], or a combination of those [28]. However, we argue instead that modules are the most suitable target artifact.

Our approach to automated testing is partly motivated by the work of Hovemeyer and Pugh [29], which states that techniques that rely heavily on formal methods or sophisticated program analysis are valuable, but difficult to apply in practice, and thus not always effective in finding real bugs. *Bug patterns* (code idioms that are often errors) can be used to facilitate the development of simple detectors (e.g., test templates) that facilitate efficient bug detection in real applications. Therefore, unlike these existing methods, we focus on *how to create tests that can detect specific bug patterns in cognitive agents*. To this end, we introduce an approach for specifying test templates that supports an agent programmer in writing tests, based on a minimal set of temporal operators that enable the specification of test conditions in a corresponding test language that we show is sufficiently expressive for detecting all failure types of an existing failure taxonomy.

3.3. Automated Testing Framework

We introduce an automated testing framework for rule-based agent programming languages that use some knowledge representation (KR) for representing the cognitive state of an agent. We aim for a framework that is as generic as possible, but in order to introduce a concrete framework we need to make a number of assumptions about these languages.

First, agents have a *perception processing component* that they use for processing received percepts, which are simple facts that we denote by $p(\vec{t})$, i.e., some identifier p with any number of arguments $\vec{t}$. We assume that percepts are stored each agent cycle when received from an environment for further processing. Second, we assume that the KR used supports a *negation* operator, denoted by `not`. Third, we assume that the cognitive state includes a *belief state* that can be queried and which allows to make percepts persistent. Fourth, we assume that agents somehow can represent *goals* that they want to achieve as part of their state; agents can do this, for example, by means of events, plans, or declarative goals. Otherwise, the structure of an agent's cognitive state depends on the specifics of the agent programming language, which almost always includes more, e.g., also events, messages, and/or plans. Fifth, we assume that the language provides support for aggregating or encapsulating basic language elements such as knowledge, goals, plans, and/or rules in what we call *modules*. Finally, we assume that an agent has a top-level module, which enables associating test conditions with the agent itself.

Three important design questions need to be addressed to define an automated testing framework for cognitive agent programs. First, we need to specify what the *basic unit or program component* is that tests are performed on or are associated

with. Second, we need to define a *test language* for specifying test conditions. And, third, we need to specify an *infrastructure that automates running tests*. In order to provide concrete examples of the concepts introduces in these three steps, we will first introduce a case study that will be used in the remainder of this chapter.

3.3.1. Case Study: Blocks World for Teams

The Blocks World for Teams (BW4T) [30] is a simulated environment in which one or more 'robots' have to work together in order to retrieve *coloured blocks* from *rooms* in the environment and deliver them to a *dropzone* in a pre-defined order. Each individual robot has to move around the environment in order to inspect the various rooms for their contents, and is able to to pick-up (and drop) one block at a time. The BW4T environment has been proposed and used as a unified testbed for multi-agent systems, and will be used as an example throughout the remainder of this chapter. To this end, in Table 3.1, we briefly introduce the concepts that can be used as *input* (i.e., percepts) for an agent controlling a BW4T robot, and the possible *output* (i.e., actions) that an agent can generate.

Percept	**Description**
$sequence(List)$	Indicates which blocks should be delivered in what order.
$in(Location)$	Indicates which location the robot is currently in.
$block(Block, Colour)$	Indicates that the robot sees a block of a certain colour.
$holding(Block)$	Indicates that the robot is currently carrying the given block.
Action	**Description**
$goTo(Location)$	Orders the robot to move to the specified location.
$pickUp(Block)$	Orders the robot to pick-up the specified block.
$putDown$	Orders the robot to put down the block that it is holding.

Table 3.1: A brief description of the percepts and actions in the BW4T environment

3.3.2. Modules as a Basic Unit for Testing

An important initial question that we need to address when setting up a testing framework is what the *target unit* for testing is in an agent-oriented program. We argue that a testing framework for agent programs should not focus on knowledge bases to avoid reinventing the wheel, but developers should rather re-use existing (unit) testing frameworks for the underlying KR technology of an agent programming language. For example, a language that uses SWI Prolog for KR should aim at re-using the available unit testing framework [31]. We have also found that, in practice, testing at the level of individual goals, plans, or rules is too fine-grained and not that useful. Writing tests at the level of individual rules, for example, would not only result in more test than source code, but even worse, would not focus on the failures that need to be detected. A more suitable level is the aggregate level that collects multiple goals, plans, and/or rules in a single unit. We call such units *modules* and assume agent programming languages provide some level of support for modules. We will therefore *associate test conditions with modules* and introduce

a test language that supports this.

Modules thus can be perceived of as components in an agent program that set goals or plans and generate actions to be executed, much like individual basic actions but at a higher level of abstraction. A module thus provides an execution unit smaller than the agent program itself but larger than other basic language elements. By using modules as targets for testing, we provide a developer with some additional control over which behaviour of the agent program is tested, as only the behaviour generated by the module will be evaluated. This allows a developer to write tests that target only specific parts of an agent program.

An agent enters a module when it starts executing the module and exits the module again when module execution is finished. These execution points provide two natural places for introducing test conditions. We will associate *pre-conditions* and *post-conditions* with a module that is evaluated, respectively, when entering and when exiting the module. Although useful, in practice pre- and post-conditions are not sufficient for monitoring the trace, i.e., behaviour and states, generated while a module is executed. To be able to evaluate a module's behaviour, we therefore also introduce so-called *in-conditions* that are associated with a module. An in-condition is a temporal property evaluated on the trace generated by a module. These conditions allow the detection of failures that occur during module execution. They also provide better support for fault localization by indicating the code location where a failure was detected, as we will explain below.

3.3.3. Test Language

A test language should provide support for two main tasks: *setting up a test* and *specifying* which *test conditions* should be evaluated. To this end, we introduce the test language in Table 3.2. A testing framework that, when provided with a *test* program as specified by the grammar, should initialize and set up the infrastructure for running an agent system and (external) environment in which the test will be automatically performed. A *time-out* can be specified to ensure termination of the test after a specified time. A time-out is global and specifies how many time (in seconds) is allowed to pass before the entire test should have been completed.

Test setup

The agents that are part of a test need to be referenced explicitly in a test program by means of their *id*'s. These agents are launched when the test is started and automatically connected to an environment, if available, to receive percepts from and perform actions in that environment. The fact that agents are launched, however, does not mean that the program code of these agents is executed. Instead, the *testaction*s that are specified in an *agenttest* clause (see Table 3.2) are performed when the test is run[1]. Test actions can be preparatory actions **do**$action$ for, e.g., initializing an agent's state, where $action$ can be a sequence of actions that are available in an agent language. Test actions can also be instructions **do**id to execute a module, with id the module's name. Note that it is possible to run an agent

[1]We note that other agents in the MAS that are not referenced in the test are launched and run as usual until all test actions have been completed.

test	:=	[*timeout*] *moduletest** *agenttest*$^+$
timeout	:=	**timeout** = *integer* **.**
agenttest	:=	*id* (, *id*)* **{** *testaction*$^+$ **}**
testaction	:=	**do**(*action* | *id*) [`until` ψ] **.**
moduletest	:=	**test** *id* **with**
		[**pre{** ψ **}**] [**in {** χ^+ **}**] [**post{** ψ **}**]
ψ	:=	Ψ | **not** (Ψ) | $\psi \wedge \psi$
χ	:=	**never** ψ | ψ **leadsto** ψ **.**
Ψ	:=	*stateop* (ϕ) | **done** (*action*)
id	:=	A simple identifier (e.g., denoting the name of an agent or module).
stateop	:=	Any state query operator (e.g., for inspecting beliefs or goals).
action	:=	Any action expression.
ϕ	:=	Any KR expression (i.e., within a state operator or action arguments).

Table 3.2: Test Language Grammar

itself by executing its top-level module (each agent is assumed to have one).

Finally, an `until` ψ condition can be associated with a module (or, less usefully, an action) that terminates execution when a state condition ψ holds (state conditions are discussed in more detail in the following subsection). An agent test thus determines which actions and modules are executed and when they should be terminated. An agent test can be shared by multiple agents, but it is also possible to define different actions for different agents, which will then be executed in parallel.

The environment that agents operate in is not explicitly referenced in our test language because the specifics of starting an external environment are very different for each agent language. Here, we simply assume that a language-specific mechanism is used for connecting agents to an environment.

Test conditions

As explained, test conditions are associated with modules. Which conditions should be evaluated when a module is executed is specified by a **test** *id* **with** statement, where *id* is a module name. With that module, a pre-condition **pre{** ψ **}**, a post-condition **post{** ψ **}**, and/or in-conditions **in {** χ^+ **}** can be associated. Any conjunction of (possibly negated) state conditions Ψ is defined as ψ; for simplicity any such combination is also referred to as 'a state condition'. Such a 'full' state condition generally describes (i.e., in terms of goals and beliefs) a *reason* for selecting a certain action. A state condition Ψ is a condition of the form $stateop$ (ϕ) on the cognitive state of an agent where ϕ is an expression in a KR language, or a clause of the form **done** (α) on an action α that an agent has just performed[2]. The state conditions that are supported will be different for each agent language, as they each use a specific cognitive state structure and associated state operators $stateop$ for inspecting that state. Querying the beliefs of an agent, for example, may be written as ϕ? in one language and as bel(ϕ) in another. Multiple state

[2]We note that because of this definition of the **done** operator, it only makes sense to use at most a single **done** clause in a state condition ψ.

conditions can be joined by a conjunction operator, which again might for example be written as ',' in a specific language, and 'AND' in another. Note that we do not introduce a disjunction operator in order to prevent complication through for example nesting conjunctions with disjunctions. Moreover, a disjunction can usually be rewritten by introducing additional tests or conditions (which is discussed below as well).

The pre-condition of a module is a state condition ψ that should hold when a module is entered (otherwise, the test fails). An example of such a condition is: goal($in(Location)$), i.e., informally the agent should have the goal to be in a certain location before entering the module this pre-condition is associated with. Similarly, a post-condition is a state condition that should hold when a module is exited (terminated). An example of such a condition is: bel($in(Location)$), i.e., informally the agent should believe it is in a certain location when leaving the module this post-condition is associated with (otherwise, the test fails).

An in-condition χ is a temporal condition that specifies which behaviour is expected of a module. Such a temporal property or condition is a statement of the form **never** ψ or ψ **leadsto** ψ'. Conditions **never** ψ can be used to specify safety conditions, i.e., things that never should occur. Conditions ψ **leadsto** ψ' can be used to specify liveness conditions, i.e., things that are supposed to occur sooner or later after something else has happened [32, 33].

Our test language only uses the two basic temporal operators **never** and **leadsto**, as these two operators turn out to be sufficient for detecting failures (as we will show). We do not want to complicate our test language more than strictly necessary, as we aim for our language to be used by trained agent programmers. We therefore want to minimize the level of acquaintance with concepts from temporal logic that is needed. In particular, similar to preventing nesting conjunctions and disjunctions, we want to avoid nesting temporal operators, as conditions would otherwise quickly become difficult to understand. A disjunction in **never** can always be rewritten to multiple **never** conditions, as well as a disjunction in the first part of ψ **leadsto** ψ'. A disjunction in ψ' can usually be resolved in a similar way by specifying more specific (strict) versions of ψ in separate **leadsto** conditions. Finally, we note that some temporal operators such as `always` ψ can be introduced as syntactic sugar for **never not**(ψ). Similarly, **eventually** ψ can be introduced as shorthand for true **leadsto** ψ.

3.3.4. Semantic model

Although it is outside the scope of this chapter to provide a formal semantics of the test conditions, we briefly introduce the basic semantic model that we assume informally. A run or trace of an agent program consists of a (finite or potentially infinite) sequence of cognitive states of the agent. Test conditions associated with a module are evaluated on (partial) traces generated by that module. For testing purposes, these conditions are assigned one of three values: *undetermined*, *passed*, or *failed*. Initially, all test conditions of a module have the value *undetermined*. The pre-condition of a module, if specified, is evaluated on the current state when entering the module and assigned *passed* when the condition succeeds, and *failed*

otherwise. Similarly, the post-condition is evaluated on the current state when a module is exited or terminated. The value of in-conditions is (re-)evaluated every time the cognitive state of the agent changes while the module is being executed. The temporal operator of the condition determines whether the value is updated:

- **never** ψ: the value is changed to *failed* if ψ holds in the state; the value is changed to *passed* if the module (or test) is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.

- ψ **leadsto** ψ': if the module (or test) is terminated, the value is changed to *passed* if every state where ψ holds has been followed by a state where ψ' holds (and vacuously so if ψ did never hold); otherwise, the value is changed to *failed*. If the module (or test) has not been terminated yet, the value is *undetermined*.

Note that when a test is terminated, all conditions will have been assigned the value *passed* or *failed*. The definition of the **leadsto** operator requires ψ' to follow *every time* that ψ held. More precisely, because the evaluation of a state condition could possibly lead to multiple solutions (e.g., when using variables), ψ' should follow for every unique solution for ψ. Take, for example, the condition:

bel($block(Block, white)$) **leadsto** bel($holding(Block)$)

This condition implies that for every white block that is believed to exist at any point in the execution of the module the condition is associated with, a holding belief should follow (before the module is left). This behaviour might be undesirable when, for example, testing the selection of a single result from a condition with multiple solutions. However, in such cases, the KR-language used can provide support. In Prolog, for example, a `setof` or similar operator can be used to produce a single solution (i.e., a list of possibilities) in ψ for which ψ' should then only hold once (i.e., for a specific or random possibility). In the previous example, if the agent's module should only collect one block out of all known white blocks, the condition could be:

bel($setof(Block, block(Block, white), Blocks)$) **leadsto**
bel($holding(AnyBlock) \wedge member(AnyBlock, Blocks)$)

Here, the $member$ predicate checks if a certain predicate is an element of a certain list (i.e., $AnyBlock$ is in $Blocks$).

Time-out mechanics

If a test is terminated because of a time-out, this does not always imply that the test is a failure; if all conditions are passed at that time, the test is considered to have passed as well. However, it is important to consider the specific nature of test conditions that have not passed (yet) when a time-out occurs, as there can be 'false positives' for ψ **leadsto** ψ' conditions, i.e., for conditions that would have succeeded if the agent had continued running after the time-out (ψ' would have been true at some later point). Naturally, such false positives have a higher chance of occurring when the expected time of seeing ψ' after having seen ψ is longer, i.e., spanning more than one evaluation of a module.

A time-out mechanism should not be implemented by simply interrupting the agent system, i.e., instantly aborting the execution of each agent (and thus test) when a time-out occurs, as this way of terminating a test will cause problems with **leadsto** conditions that will hold within a single evaluation of a module; an interruption might then occur at any point in the module's execution. A better time-out mechanism that prevents false positives from these 'short' **leadsto** conditions can be implemented by 'evaluating the time-out' at fixed points in the execution, e.g., at the end of each module evaluation. Such a mechanism will prevent false positives in for example a module that processes percepts; if such a module with multiple decision rules that process percepts into appropriate beliefs is aborted before all rules have been evaluated, any related **leadsto** conditions for percepts handled by rules after the point of interruption will fail.

It is in any case important for a developer to consider that the result of a test condition that either held (i.e., **never** ψ) or did not hold (i.e., ψ **leadsto** ψ') when a time-out occurs might have changed when the program had run for a longer time. It should thus be clearly stated in test results if a test condition failed before or after a time-out occurred.

Based on the testing framework as introduced in this section, we will introduce test templates and a corresponding test approach in the following section.

3.4. Testing for Failures

In this section, we provide a systematic approach for detecting failures by introducing test templates that target specific types of failures in the taxonomy of Winikoff [8], given in Table 3.3, and a systematic method for using these templates. The introduced test templates can be considered a refinement of the failure taxonomy.

P1:	failure to deal with percept
P2:	other incorrect percept processing
G1:	failure to add a goal that should be added
G2:	failure to drop a goal that should be dropped
G3:	adding a goal that shouldn't be added
G4:	incorrectly adding a second goal of the same type
G5:	dropping a goal that shouldn't be dropped
A1:	selecting the wrong (user-defined) action
A2:	beliefs not updated correctly when action performed
A3:	action selected when should be doing nothing
A4:	action interface mismatch
O:	other failure not classified above

Table 3.3: Failure Taxonomy of Winikoff [8]

3.4.1. Test Templates

A test template consists of one or more templates for individual test conditions. We provide test templates for each failure type in the taxonomy, except for *A4* (action interface mismatch), which calls for another detection method and raises a specific design issue for testing frameworks. We note that a test condition specifies expected behaviour, and a violation indicates a failure. An example is provided with each template to illustrate this.

P1, P2: Failure to process percept (correctly) In order to define test conditions for percept processing, we assume a state operator percept($p(\vec{t})$), indicating that a percept named p with arguments $\vec{t}$ is perceived, that can be used to inspect the contents of the stored percepts. This does not mean that these conditions must be supported in the programming language itself, but only that it should be possible to somehow inspect which percepts have been received.

In order to support various options for percept processing, we distinguish three ways in which a percept can be updated, and associate specific test templates with each such method. Although in theory alternatives can be conceived of, we have specified test templates below based on the assumption that the percept information needs to be made persistent in the agent's belief state, which worked well in practice. We also assume that test conditions for percepts can be associated with a percept processing module (abbreviated *ppm*), and we can write test statements of the form **test** *ppm* **with in {** χ^+ **}**. Each agent language provides some kind of support for this, although perhaps not in the language itself but 'under the hood'. It is important that test conditions for percepts are associated with and evaluated while executing the percept processing module. Because such a module is generally executed once per agent decision cycle, in order to not violate the test conditions, percepts must have been processed and beliefs updated accordingly at the end of percept processing.

Template *P-once*: concerns percepts that are only received *once*, typically when the agent is launched to inform it about static information such as locations on maps. The test template expects that after receiving the percept, it will be made persistent in the agent's beliefs.

> percept($p(\vec{t})$) **leadsto** bel($p(\vec{t})$)

For example, the condition 'percept($sequence(List)$) **leadsto** bel($sequence(List)$)' verifies if the sequence of coloured blocks that is sent only when the agent starts is correctly inserted into the agent's beliefs, i.e., that the percept has been (correctly) processed into a belief the end of the percept processing module with which the condition is associated.

Template *P-always*: concerns percepts about facts $p(\vec{t})$ that are *always* received when $p(\vec{t})$ is the case, meaning that not receiving the percept implies that $p(\vec{t})$ does not hold. For example, a percept $block(1, red)$ is sent to an agent as long as it can see that red block. The corresponding test template consists of two test

conditions. The first is identical to the one for ***P-once***; the second condition expects that not perceiving $p(\vec{t})$, which would indicate that $p(\vec{t})$ does not hold, should lead to removing the belief $p(\vec{t})$ (if present).

percept($p(\vec{t})$) **leadsto** bel($p(\vec{t})$)
(**not**(percept($p(\vec{t})$)) ∧ bel($p(\vec{t})$)) **leadsto** **not**(bel($p(\vec{t})$))

The conditions 'percept($block(Block, Colour)$) **leadsto** bel($block(Block, Colour)$)' and '(**not**(percept($block(Block, Colour)$)) ∧ bel($block(Block, Colour)$)) **leadsto** **not**(bel($block(Block, Colour)$))' can for example be used to verify that the agent's beliefs about the blocks that it can see are correct.

Template *P-on-change*: concerns percepts that are sent only when parameters change. The percept $in(Location)$, for example, is sent each time an agent's location changes.

percept($p(\vec{t})$) **leadsto** bel($p(\vec{t})$)
(percept($p(\vec{t})$) ∧ bel($p(\vec{t'})$) ∧ $\vec{t} \neq \vec{t'}$) **leadsto** **not**(bel($p(\vec{t'})$))

Here, the conditions 'percept($in(Location)$) **leadsto** bel($in(Location)$)' and '(percept($in(Location)$) ∧ bel($in(Current)$) ∧ $Location \neq Current$) **leadsto** **not**(bel($in(Current)$))' illustrate verifying the agent's beliefs about its location.

The three test templates for percepts assume that any percept information should be stored in the agent's beliefs 'as is'. In practice, information contained in a percept may be purposely left out of the beliefs, or information from multiple percepts may be combined. The generic structure of the above templates still applies in such situations depending on the type of percept updating that is applicable, but some instantiations of $p(\vec{t})$ will need to be changed depending on the specific situation. Moreover, in some situations, removing a belief when something is (perhaps temporarily) no longer percepted might not be desired in the specific implementation, in which case the corresponding test condition can be left out.

G1: Failure to add a goal that should be added The taxonomy in Table 3.3 includes five failure categories related to goal handling. We therefore assume that a state operator goal is available for checking for the presence or absence of a goal. Each of these failure categories, with the exception of *G4*, suggest that a *reason* for (not) having a goal has not been adequately taken into account.

Template *G-adopted*: concerns a goal ϕ that the agent is expected to adopt because of some (sufficient) reason ψ.

ψ **leadsto** goal(ϕ)

For example, the functional requirement that an agent adopts an 'in-goal' for every unvisited room that it beliefs to exist can be formalized as a condition
'(bel($room(Loc)$) ∧ **not**(bel($visited(Loc)$))) **leadsto** goal($in(Loc)$)'.

G2: Failure to drop a goal that should be dropped The failure to drop a goal is taken here as an indication that the agent did not adequately reconsider the goals that it has. As one would expect an agent to reconsider its goals if the environment has changed outside the control of that agent, these failures would most likely only occur in dynamic environments or in a multi-agent context.

Template *G-reconsider*: concerns a goal ϕ that should be reconsidered and dropped as a result for some reason ψ.

ψ **leadsto not**(goal(ϕ))

The condition '(goal($holding(Block)$) ∧ **not**(bel($needBlock(Block)$))) **leadsto** **not**(goal($holding(Block)$))' illustrates this failure type, where $needBlock(Block)$ is some predicate that checks if $Block$ needs to be delivered according to the goal sequence, which might not be the case any longer when another agent in the environment delivered a block.

G3: Adding a goal that should not be added *G3* is the counterpart of *G1*, reflected by the fact that we use a safety (**never**) instead of liveness (**leadsto**) condition here.

Template *G-incorrect*: concerns a situation in which there is a reason ψ for not adopting (having adopted) the goal ϕ.

never(goal(ϕ) ∧ ψ)

The condition '**never**(goal($in(Location)$) ∧ bel($visited(Location)$))' can for example be used to verify that the agent never adopts a goal to visit a location that has been visited before.

G4: Incorrectly adding a second goal of the same type Some goals should only occur once and it should never be the case that the goal is instantiated twice. For example, an agent might have a goal of visiting a certain location but should never have two of those goals simultaneously.

Template *G-duplicate*: concerns a single-instance goal $\phi(\vec{t})$ that should be instantiated at most once.

never (goal($\phi(\vec{t})$) ∧ goal($\phi(\vec{t'})$) ∧ $\vec{t} \neq \vec{t'}$)

Verifying that the agent has at most one 'holding-goal' can for example be done with the condition '**never** (goal($holding(Block)$) ∧ goal($holding(Other)$) ∧ $Block \neq Other$)'.

G5: Dropping a goal that should not be dropped Similar as *G3* is to *G1*, *G5* is the counterpart of *G2*.

Template *G-maintain*: concerns a situation in which ψ is a reason why an agent should have a goal ϕ, and should maintain it for that reason.

never (**not**(goal(ϕ)) $\wedge$ ψ)

The condition '**never** (**not**(goal($in('DropZone')$)) $\wedge$ bel($holding(Block)$))' can for example be used to verify that the agent always has the goal to be in the dropzone whilst it is holding a block (i.e., in order to deliver the block there).

A1: Failure to select an action Table 3.3 includes four failure categories related to actions. Categories *A1* and *A3* suggest that a *reason* for (not) selecting an action has not been adequately taken into account.

Template *A-selected*: concerns an action α that the agent is expected to have selected because of some (sufficient) reason ψ.

ψ **leadsto done**(α)

An illustration of this template is the condition '(bel($holding(Block), in('DropZone')$)) **leadsto done**($putDown$)', which can be used to verify that the agent puts down any block that it is holding when located in the dropzone.

A2, A3: Incorrect action (selection) We provide one test template for categories $A2$ and $A3$, which both suggest that something happened that should (never) have happened, and thus can be viewed as the counterpart of $A1$. $A2$ indicates that beliefs should not have been updated the way they are, and $A3$ indicates there is a situation in which an action should never have been selected.

Template *A-incorrect*: concerns an action α that should never be (immediately) followed by ψ ($A2$), or a situation ψ in which an action should never have been selected ($A3$).

never (**done**(α) $\wedge$ ψ)

For example, the condition '**never** (**done**($putDown$) $\wedge$ **not**(bel($in('DropZone')$)))' can be used to verify that the agent will never put down a block outside of the dropzone.

A4: Action interface mismatch This failure concerns an agent that attempts to perform an action in an environment that is not supported by that environment. An environment is expected to raise an exception or use another event-mechanism to indicate this issue. To detect these failures, rather than providing a test template, the testing framework should abort testing immediately after receiving the exception or event. Aborting upon receiving an exception is a general principle that is supported in our framework.

3.4.2. Taxonomy Refinement

The test templates that we have introduced, as summarized in Table 3.4, do not offer a perfect match with the failure categories, but provide a refinement of the taxonomy in Table 3.3. Instead of two categories *P1,P2* for percept processing failures, we introduced four templates (***P-once***, etc.). The main reason for this difference is that in a test approach we should specify what kind of percept is expected, instead of indicating that no processing was done at all (*P1*) or something went wrong (*P2*). For a similar reason, we renamed *A1* from 'selecting the wrong action' to 'failure to select an action', which also highlights the similarity with the template for *G1*. We merged *A2* and *A3* which are covered by a single ***A-incorrect*** test template; note again the structural similarity with the template for ***G-incorrect*** (*G3*). As discussed, we do not have a corresponding template for *A4*, as tests should rather be aborted when this failure happens. Finally, we have no templates for the 'other' category, as our empirical results suggest there is no need for an additional template.

3

P-once:	failure to process a percept that was received only once
P-always:	failure to process a percept that is always received when it holds
P-on-change:	failure to process a percept that is received when parameters change
G-adopted:	failure to adopt a goal that should have been adopted
G-reconsider:	failure to drop a goal that should have been dropped
G-incorrect:	incorrectly adopting a goal that should not have been adopted
G-duplicate:	incorrectly adopting a second instance of a single-instance goal
G-maintain:	incorrectly dropping a goal that should not have been dropped
A-selected:	failure to select an action that should have been selected
A-incorrect:	incorrectly selecting an action that should not have been selected

Table 3.4: Test templates indicating a refined failure taxonomy

3.4.3. Test Approach

In order to test and use the test templates, we need a *systematic test approach* that tells us what to do (steps) and, more specifically, provides clues on how to instantiate the templates for a specific application. An important question, for example, is how to find reasons to instantiate the ψ conditions that occur in the ***G***- and ***A***-templates. Table 3.5 lists information resources that are particularly useful for testing.

Step 1: defining success

The first step is to identify functional requirements from available agent design documentation (Table 3.5). These requirements define success and provide a concrete method for checking that a program does what it is supposed to do. A program can be considered free of failures if it meets all its requirements. In order to automatically check this, functional requirements must also be specified in the test language.

Source	Type of Information
Agent program (comments)	Clues for reasons & design
Agent trace (screen, logs)	Observable behaviour
Agent design & specification	Functional requirements
Environment (documentation)	Percepts, actions available

Table 3.5: Information sources for testing

3

Typically, these requirements will be associated with the top-level module, called `main` here, that is executed by the agent, and we can specify them as pre-, post-, or in-conditions of that module using **test** `main` **with** statements or by using a statement **do**`main until` ψ. The latter is particularly useful for checking that some overall objective ψ is realized (and, if so, the test will be automatically terminated; a `timeout` should be specified to guarantee termination if this is not the case). For example, the requirement or objective to pickup and deliver n blocks can be specified by **do**`main until` bel($delivered(n)$).

Step 2: testing cognitive state updating

It is important to test that updating of an agent's cognitive state works as expected. The state conditions used in test conditions are evaluated on this state and will fail for unclear reasons when state updating has not been implemented correctly. For example, a condition **never**(**done**($putDown$) $\wedge$ **not**(bel($in('DropZone')$))), which expresses that a block should never be put down when not in the dropzone, could simply fail because beliefs about $in(Location)$ were not updated correctly.

Identify percepts, actions, and goals used As a preparatory step, it is useful to collect and list all types of percepts that can be generated by the environment, i.e., a percept's name, (number of) arguments, and update type (*once*, etc.). Similarly, the types of actions that may be performed in the environment should be collected, and all types of goals that an agent may have should be collected from the agent program code (all by their name and possible arguments).

Validating percept processing The most basic step is to instantiate the appropriate test templates ***P-once***, etc. for each percept according to its update type. The resulting test conditions should be associated as in-conditions with the percept processing module, and the testing framework used to evaluate these conditions. Tests should be repeated sufficiently often as percepts generated will differ per run, if only because environments are more often than not non-deterministic. To gain confidence that percepts are correctly handled, it is important to (manually) check against the list of actions created above whether a sufficient variation of actions has been performed during test runs, as different actions often yield other percepts.

Check single-instance goals Based on the design and intended use of goals in a program (annotated by comments for example, see also Table 3.5), and using the goal list created in Step 1, the subset of goal types that are single-instance

goals should now be identified. For each of these, the test template ***G-duplicate*** should be instantiated and associated as an in-condition with the module(s) where a corresponding goal is adopted.

We note that the first and second steps can be performed by almost mechanically instantiating templates once relevant information has been collected. If these initial tests succeed, therefore, this will give a high level of confidence that cognitive states are updated correctly. For all templates other than those used up until now, however, a state condition ψ needs to be derived using insights gained from the design and behaviour of an agent.

Step 3: classifying failures

After testing state updating and checking that failures still are present, there are two approaches for classifying those failures. A *bottom-up* approach would start with the action and goal list created and try to instantiate templates for all these actions and goals, using the agent program itself as the main source of information. A *top-down* approach would rather start by analysing agent traces (see also Table 3.5) and observing agent behaviour in order to identify which action should (not) have been performed or goal should (not) have been added. Although a top-down approach suggests to start with testing goals, this order is not fixed and testing might just as well proceed with actions.

Failures concerning actions In order to instantiate ***A***-templates, apart from the action, a state condition ψ should be identified that provides a reason for (not) selecting an action. For ***A-selected*** (resp. ***A-incorrect***), the question is in which situations an action should (never) be executed. The instantiated conditions should be associated as in-conditions with the module(s) where the action might (not) be selected.

Bottom-up approach

By inspecting the agent program, clues may be obtained for useful test conditions ψ. In particular, the triggering conditions of rules can be useful, as they typically indicate reasons for selecting an action. For example, a (rule) condition bel($in('DropZone') \wedge holding(Block)$) that triggers execution of an action $putDown$ suggests that an agent should execute $putDown$ when it is holding a block in the $'DropZone'$. By simply using this condition for ψ we can instantiate ***A-selected*** as follows:

(bel($in('DropZone') \wedge holding(Block)$)) **leadsto done**($putDown$)

This simple approach can already detect failures, e.g., in case a wrong ordering of rules prevents the rule for $putDown$ to ever be applied. Similarly, by simply negating conditions found in a program, we can instantiate ***A-incorrect***. This assumes that the condition must hold for the action to be selected. Although these assumptions may be too strong, they provide a useful starting point and can be weakened based on further analysis.

Top-down approach

If an action failure is suspected because, for example, a functional requirement is not satisfied, observing an agent's behaviour may provide clues for identifying a

useful condition ψ for instantiating a test template for that action. Suppose the following condition fails:

bel($in(Loc)$) **leadsto not**(bel($in(Loc)$))

This indicates that the agent does not always leave a location after entering it. By identifying that the *goTo* action is required to leave a location and observing that it is never performed, a failure to select that action is suggested. To test for this, the ***A-selected*** template can be instantiated by instantiating ψ with the reason for leaving and α with the *goTo* action, which yields:

(bel($in(Loc) \wedge Loc \neq OtherLoc$)) **leadsto done**($goTo(OtherLoc)$)

This process can be continued until the root cause has been identified.

Failures concerning goals The approach for instantiating ***G***-templates, apart from identifying the goal that might cause the failure, is similar to that for ***A***-templates. The associated questions for the templates are, for ***G-adopted***: for which ψ should the goal be added?; for ***G-reconsider***: for which ψ should the goal be dropped?; ***G-incorrect***: for which ψ should a goal never be added?; and for ***G-maintain***: for which ψ should a goal never be removed? The approach for identifying ψ for actions above also applies for goals. The instantiated conditions should be associated as in-conditions with the module(s) that are related to the goal. We illustrate a test for a goal $in(Loc)$ that is adopted in a rule with a triggering condition bel($room(Loc)$) $\wedge$ **not**(bel($visited(Loc)$)), i.e., indicating that an agent should adopt (multiple) goals to be in a certain location when it knows about a room that it has not visited before. By simply using this condition for ψ we can instantiate ***G-adopted*** as follows:

(bel($room(Loc)$) $\wedge$ **not**(bel($visited(Loc)$))) **leadsto** goal($in(Loc)$)

Again, this simple approach can already detect failures, e.g., in case the rule order prevents the rule from ever being applied. Similarly, by simply negating conditions found in a program, we can instantiate ***G-incorrect***. This assumes that the condition must hold for the goal to be adopted, e.g., we never want to go to rooms we have visited before. A similar approach can be used for the ***G-reconsider*** and the ***G-maintain*** templates.

3.4.4. Test Results

In order to facilitate fault localization, it is important to consider how the results of a test are presented to a developer. First, we will discuss what the (textual) results of an agent test should look like when not considering integration into a source-level debugger. Next, we will discuss how tests can be integrated into a source-level debugger.

Output

The results of a test can be displayed by printing the evaluations of the test conditions (to some console) when all the actions in a test have been completed (or a time-out occurs). Alternatively, one could consider running a test until the first failure occurs, and printing only that failure. Both could be supported by adding a

toggle that enables either one or the other mode of running. First and foremost, the user should be notified if the test has passed overall, i.e., if all conditions held. In that case, just printing this is sufficient. However, more interestingly, when a test fails, the user should be informed about the specific test conditions that have failed (per module).

A condition of the form **never** ψ can only fail at one specific point in a module's execution, i.e., in the first cognitive state that ψ held. Therefore, the exact details of ψ should be displayed, i.e., the instantiation (variables) that made the condition hold. Moreover, it can be useful to provide some information about the point in time at which the condition held, for example by adding the number of decision cycles that an agent had performed up until that point to the result. This can for instance be used to look up the mentioned cycle in agent logs, and from there search for further information. An example of the result of a **never** condition would be:

> The condition **never** (**done**($goTo(Location)$) $\wedge$ bel($visited(Location)$)) failed in cycle 51 with *Location='Room1'*.

A condition of the form ψ **leadsto** ψ' can only fail at one specific point in a module's execution, i.e., when the module exits and ψ' did not hold yet. However, multiple specific instantiations of ψ can require multiple specific instantiations of ψ' to hold. It is even possible that specific instantiations of both ψ and ψ' already held at some point in the module's execution, but not for the final time that ψ was seen, as each time that ψ holds ψ' should follow at some later point in the module's execution. Therefore, not only should the exact details of ψ be displayed, i.e., the instantiation (variables) for which the condition held, but also the exact details of the possible multiple instantiations of ψ' that did not hold. Again, providing information about the timing of such failures can be useful, i.e., adding the number of decision cycles that the agent had performed up until ψ held (from which ψ' did not follow). Moreover, because these conditions can fail due to a time-out (and perhaps falsely so), the result should clearly state if the specific failure occurred before or after the time-out. An example of such a result would be:

> The condition **not**(bel($visited(Location)$)) **leadsto** **done**($goTo(Location)$) failed as when the first part of the condition held in cycle 151 with *[Location='Room1',Location='Room2']*, the second part of the condition did not hold with *Location='Room2'* before the module exited in cycle 152.

Another variation that should be taken into account is when a condition ψ **leadsto** ψ' did hold, but vacuously so, i.e., when ψ was never satisfied.

We note that it is also possible some (run-time) exception occurs during the execution of a test action. In this case, the test itself should be regarded as failed, and the exception printed in the test results.

Debugger integration

Besides using a console (for logging output), fault localization can be facilitated even more by integrating a test in a source-level debugger (c.f. Chapter 2). Such a debugger allows single-step execution of a program, providing a better insight into the relationship between program code and the resulting behaviour. When executing an agent test, such a debugger should suspend the execution of an agent

whenever a test failure occurs in order to allow a developer to inspect the agent's cognitive state at that exact point (and perhaps its log up until that point). When this happens, the details of the failure should be printed as well, as discussed in the previous part. The failing test condition combined with its specific location in the agent's source code and the agent's current cognitive state will provide a developer with strong clues about which parts of the agent to inspect. For example, when a condition **never** $\big(\, \textbf{done}(goTo(Location)) \wedge \text{bel}(visited(Location)) \,\big)$ fails (i.e., directly after $goTo(Location)$ was executed for an already $visited$ location), the source-level debugger should suspend the agent and highlight the location at which the failure occurred, which is in this case a call to $goTo(Location)$ in some specific decision rule. The condition of that rule might already be the case of that failure, when for example it was not strong enough, or otherwise it will provide further clues towards the fault location, when for example a certain goal lead to the action being performed (thus spawning further investigation into that goal).

When a failure occurred during debugging, the developer can either choose to halt and correct the agent at that point, or continue running (or perhaps single stepping) until either the next failure occurs or the test ends.

3.5. Testing GOAL Agents in the Eclipse IDE

A concrete implementation of the proposed automated testing framework has been performed for the GOAL agent programming language [34], embedded in its plug-in[3] for Eclipse. This plug-in provides a full-fledged development environment for agent programmers, integrating all agent and agent-environment development tools in a single well-established setting [35]. A source-level debugger for agents has been integrated in this plug-in (c.f. Chapter 2). Full editing support has been added for test files (with extension `.test2g`), as illustrated in Figure 3.1.

The testing framework has been fully integrated into the source-level debugger, as illustrated in Figure 3.2. In the plug-in there are several examples embedded that can be used to demonstrate the testing framework. Most of these are based on educational environments[4] that usually include an assignment for (novice) agent programmers.

3.5.1. Implementation Details

As detailed in Chapter 2, a listener can be attached to the GOAL runtime, which emits specific events (i.e., the breakpoints). The testing framework uses this same mechanism in order to determine when to (re-)evaluate test conditions. First, when a module is entered, its precondition is evaluated, and any test conditions associated with it are added to the general set of test conditions to evaluate for that specific agent. Then, upon each change to the agent's cognitive state (e.g., a belief is inserted or a goal is achieved), all test conditions in this set are evaluated. Conditions of the form **never** ψ remain in this set until the module is exited. For

[3] See the tutorials at https://goalapl.atlassian.net/wiki for a demonstration(video) of the testing framework implementation and instructions on how to install GOAL in Eclipse.

[4] All (educational) agent environments are available at https://github.com/eishub/.

vacuum.test2g

```
use vacuumWorld as mas.
use vacuum as knowledge.
use vacuumInit as module.
use vacuumMain as module.
use vacuumActions as actionspec.
timeout=20. %seconds

test vacuumInit with
    post{ bel(cleaned(0)), goal(cleaned(8)) }

test vacuumMain with
    in{
        % cleaned/1 goal
        never goal(cleaned(X)), goal(cleaned(Y)), not(bel(X=Y)).

        % location/1 goal
        never goal(location(X,Y)), not(bel(square(X,Y,dust))).
        never goal(location(X1,Y1)), goal(location(X2,Y2)), not(bel(X1=X2,Y1=Y2)).

        % move/1 action
        goal(location(X,Y)), bel(goto(X,Y,Dir)) leadsto done(move(Dir)).
        never done(move(Dir)), goal(location(X,Y)), not(bel(goto(X,Y,Dir))).

        % clean/0 action
        bel(square(here,dust)) leadsto done(clean).
        never done(clean), not(bel(square(here,dust))).
    }

vacuumbot {
    do vacuumMain until bel(cleaned(8)).
}
```

Figure 3.1: An example of a `test2g` file being edited in the GOAL plug-in for Eclipse.

conditions of the form ψ **leadsto** ψ', initially only ψ will be in the set of test conditions for the agent. Then, for each time ψ holds (i.e., after a single change to the agent's cognitive state), all applicable variations of ψ' will be added to the set of conditions (i.e., if the specific instantiation is not already present). In turn, when any specific instantiation of ψ' is already in the set, whenever it holds (i.e., again after any change to the agent's cognitive state) it is removed from the set. Then, in the end when the module is exited, at which point its postcondition will be evaluated, each remaining condition ψ' in the set indicates a specific failure of the complete ψ **leadsto** ψ' condition.

Test failures behave in a similar way as user-defined breakpoints (c.f. Chapter 2), i.e., always halting the execution of an agent immediately when they occur.

3.6. Evaluation

In this section, we first evaluate whether our automated approach is able to detect all failures as previously identified in a specific set of agent programs [8]. Next, we evaluate whether we are able to detect failures in a different set of agents operating in the same environment, but with additional functional requirements.

```
44     if bel(atBlock(X)), percept(not(atBlock(X))) then insert(not(atBlock(X))).
45
46
47
48
```

Console robot

robot

```
exception: The condition 'never goal(holding(X)), not(bel(atBlock(X)))' did not hold; the evaluation [[X/56]] applies.
```

Figure 3.2: An example of how test failures are displayed in the source-level debugger. Information about the agent's cognitive state at this point is available elsewhere in the debugger.

In addition, we evaluate this for a single agent program sample that operates in a different environment. Next, we evaluate the effort that is required to reproduce failures in both single and multi agent systems. Finally, we discuss both empirical and qualitative evaluations that have been performed with novice programmers who have used the testing framework for GOAL.

3.6.1. Comparing the Approaches

To establish the effectiveness of our automated testing framework and test approach, we evaluate here whether the automated testing framework, using the test templates introduced above, is able to detect the failures that were detected by manual inspection by Winikoff [8]. We aim to show that our framework can be used to detect failures to obtain failure-free programs, meaning that functional requirements are met.

Methodology

We used the agent program sample and the methodology for testing, debugging, and classifying failures of Figure 3.3 that was also used by Winikoff [8]. Only step 1 and 3 were replaced by tests performed using our testing framework instead of manual inspection of logs and traces. We also used the same 'success criteria', i.e., the requirement formulated at the end of Step 1 of our approach. As we want to demonstrate that failures detected by Winikoff [8] can also be detected automatically, we used a bottom-up approach for writing tests and created test templates for all percepts, actions, and goals in a program to ensure as large a coverage as possible. Upon detecting a failure (bug), we applied the same fixes as Winikoff [8], which we obtained from the author, and re-ran our tests. We verified that the same bug was found by our tests, e.g., by checking that the code location where the test was terminated corresponds with the code location where a fix was applied. If we could not match a failure found by automated testing with one found originally by manual inspection, this failure was registered separately, the corresponding test conditions were removed, and the evaluation was continued.

Results

We analysed and wrote a large set of test conditions for 20 agent programs. The results in terms of number of failures found are summarized in Table 3.6. Each individual failure found was counted and included in the table. We used the taxonomy of Table 3.3 for classification, and not our refinement, to allow for comparison.

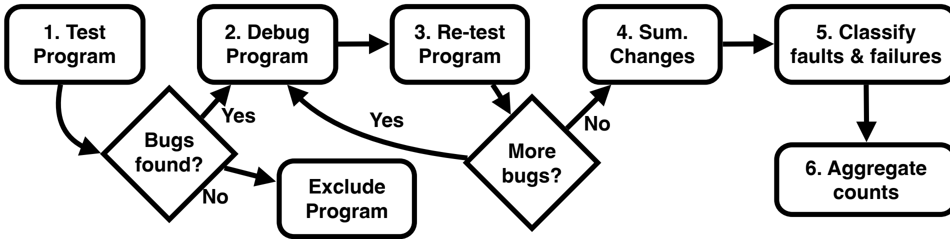


Figure 3.3: The methodology used in Winikoff [8]

3

We were able to reproduce every failure found by Winikoff [8] ('Manual' column in Table 3.6) and often found more ('Difference' column). In particular, we were able to automatically verify that all programs were failure-free after fixing buggy programs by testing that they met the requirement, indicating success.

Failure	Manual [8]	Automated	Difference
P1	14	17 (3)	3
P2	11	16 (1)	5
A1	29	30 (5)	1
A2	3	4 (0)	1
A3	4	5 (0)	1
A4	1	1 (0)	0
G1	5	5 (0)	0
G2	3	3 (0)	0
G3	7	10 (1)	3
G4	5	11 (0)	6
G5	0	0 (0)	0
Totals	82	102 (10)	20

Table 3.6: Comparison of results (6.1)

We were initially able to detect 88% of the original failures by performing automated tests. So even though we were able to fully automatically establish that a program was failure-free, detecting some problems that were originally classified as failures required manual inspection (counts between brackets in Table 3.6). After analysis, it turned out that all these failures only occur when agent's cycles are delayed indefinitely while an environment may continue running. This can happen when a developer manually pauses an agent. As an example, an agent that runs at normal speed and is connected to the BW4T environment, which we used in our sample, would never simultaneously receive the percepts *in('Room1')* and *in('Room2')*. This is because it takes time to move from *Room1* to *Room2*, and an agent running at normal speed would first receive the first and only later receive the second percept. If an agent is paused, because the environment continues running both percepts may be queued and received simultaneously when the agent is resumed. When we run tests with agents using the automated testing framework, we do not detect

such 'failures' because agents are never paused. Our finding thus highlights that it is important to take timing issues into account. We eventually reproduced 100% of the original failures by introducing (manual) pauses to our tests. We note that no *G5* failures were found (confirmed after discussion with the author of Winikoff [8]), but we did find such a failure in the sample we discuss in the next section.

We identified 24% more failures, most of which (70%) were detected by means of the percept test templates (*P1*, *P2*) and the template ***G-duplicate*** (*G4*). Interestingly, these templates are used in Step 2 of our test approach. This suggests that manual inspection is biased more towards a top-down approach in Step 3. As we argued, developing and testing a program is facilitated, however, if confidence is gained first that state updating is free of failures. One likely other reason that explains why we found more failures is that we were able to perform many test runs, as the testing framework facilitates performing repeated test runs, and each run may produce different failures because of non-determinism.

A particularly interesting result is that we found that 63 from the 82 (77%) failures detected by automated tests immediately pointed at the code location of the fault. This, of course, makes it easier for a developer to fix a bug, and is a clear indication of the value that automating testing can have for agent programming. Fault locations are hardly ever pointed at, however, when a wrong action is performed (***A-selected***, *A1*), suggesting that locating faults for such failures requires a different approach.

We now show that our approach is also able to adequately detect failures in sample programs other than those used to reproduce the results of Winikoff [8]. The test templates that we proposed have been based on the failure taxonomy of Winikoff [8] which was derived from an analysis of the sample used in the previous section.

3.6.2. Different Agent Program Sample

We aim to provide additional support that shows that our set of test templates is complete and our approach is not biased towards a particular sample by looking at other sample programs. We therefore selected a new sample of 10 agent programs that were written for the BW4T environment but by different programmers. Moreover, these programs were supposed to satisfy several more functional requirements instead of only one. Agents were required, for example, to not do any redundant tasks. For example, the requirement "An agent should go directly to the drop zone when it is holding a block" was formalized as:

$$\textbf{never}\ (\ \textbf{done}(goTo(Location)) \wedge Location \neq 'DropZone' \wedge \text{bel}(holding(Block))\ )$$

Another example is the requirement "An agent should not go to the drop zone without holding a block", formalized as:

$$\textbf{never}\ (\ \textbf{done}(goTo('DropZone')) \wedge \textbf{not}(\text{bel}(holding(Block)))\ )$$

These additional requirements pose a greater challenge for our test approach, as a program is considered failure-free only if all requirements are satisfied.

All failures found could be classified using our refined taxonomy; Table 3.7 presents counts per type. We did not find any *A4* failures but did find a ***G-maintain***

(*G5*) failure. Failures that were detected were fixed using the same approach as Winikoff [8]. We thus were able to successfully generate failure-free agents that satisfy all functional requirements.

Failure	Count
P-failure (P1,P2)	10
A-selected (A1)	11
A-incorrect (A2,3)	7
G-adopted (G1)	4
G-reconsider (G2)	1
G-incorrect (G3)	16
G-duplicate (G4)	3
G-maintain (G5)	1
Total	53

Table 3.7: Results of the performed evaluation (6.2)

A larger set of functional requirements facilitated the use of a top-down approach rather than a bottom-up approach, as was used for reproduction. An interesting finding is that this lead to a decrease in the amount of test failures that pointed directly at code locations of corresponding faults (25%, 13 out of 53, as opposed to 77% before). More work is needed to explain this finding.

3.6.3. Different Environment

We also performed an evaluation on whether the testing framework and approach would detect failures in a single agent program sample that operates in a different environment. We chose an environment called the Vacuum World to this end [36]. In this world, squares in a grid can be either clean, dusty, or contain an obstacle that the robot should move around, thus requiring the agent to also successfully navigate the robot over the grid. A robot is able to move to any neighbouring clean or dusty square and clean up (vacuum) the square it is currently on (removing the dust).

For the evaluation, we used an agent program sample that was developed to meet the requirement that all dusty squares in a grid have been cleaned. This initial agent program did not meet the functional requirement specified, and we thus applied our test approach to detect failures.

Four failures were detected in the agent program, which is similar to the average of about four or five failures that were found for all other sample programs. Two percept failures were detected using the ***P*** templates, and two failures related to action selection were detected using the ***A-selected*** template. After fixes were applied for these failures, the agent did meet its requirement. Consistent with earlier results, percept failures indicated code locations for corresponding faults, whereas the action failures required more debugging effort for fault localization.

3.6.4. Reproduction

In this part, we empirically investigate and compare the reproduction rate of failures in an agent (system). In other words, we determine how often an agent system has to be run on average to reproduce a specific failure, which is an indication of the effort required for testing (single or multi) agent systems in general.

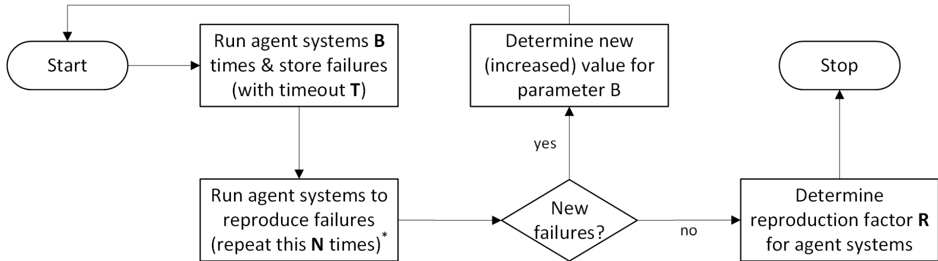


Figure 3.4: A flowchart of our evaluation method for the test reproduction rate.

Based on Chapter 2, the method we used is to first run a given test set many times in order to identify all failures in an agent system. We then ran multiple sessions in which those same tests were repeated in order to determine how many repetitions are needed (on average per session) to reproduce all the failures that were initially found. We used a time-out parameter as agent systems may run indefinitely without making any progress (i.e., producing no new test results). Failures that are due to a time-out are ignored because these cannot be consistently reproduced. Our method, as illustrated in Figure 3.4, uses the following parameters:

- B: number of initial test runs for each agent system.
- T: number of seconds after which a test is aborted.
- N: number of 'sessions' for each agent system.
- M: maximum number of test runs in a single session.

The aim is to empirically determine the reproduction factor R: the number of times a run needs to be repeated to detect all failures in an agent system.

We used two sets of agent systems programmed in GOAL that control robots in the BW4T environment. They were created by pairs of first-year Computer Science bachelor students and handed-in with accompanying tests (i.e., `test2g` files). The first set consists of 84 single-agent systems, and the second set of 42 multi-agent systems (3 agents). We applied our method to both sets of agents, with parameter B set to 100, T to 60 seconds, N to 10, and M to 1000. These parameters were chosen after an iterative process of running experiments to minimize the runtime whilst making sure that unreproducible nor new failures would be found in the repetition part of our method (see also the step to increase parameter B in Figure 3.4). In total, for our final experiment, almost 23,000 runs were performed with a total runtime of about 330 hours.

We found a significantly lower number of failures for the single-agent systems (on average 0.3 failures, with a maximum of 8 failures) than for the multi-agent systems (on average 4.3 failures, with a maximum of 21 failures). In a rather static environment like BW4T with a low number of failures, we found that a single agent's failure set can be reproduced on every single run, i.e., $R = 1$. This is very different for multi-agent systems where on average $R = 11$ runs were needed to reproduce all failures that were initially found. The distribution of R for the multi-agent systems is shown in Figure 3.5. Even though on average only 11 runs were needed, and the majority of failures only needed 1 run to be reproduced, in a few cases up to even 300 repetitions were needed to reproduce all failures.

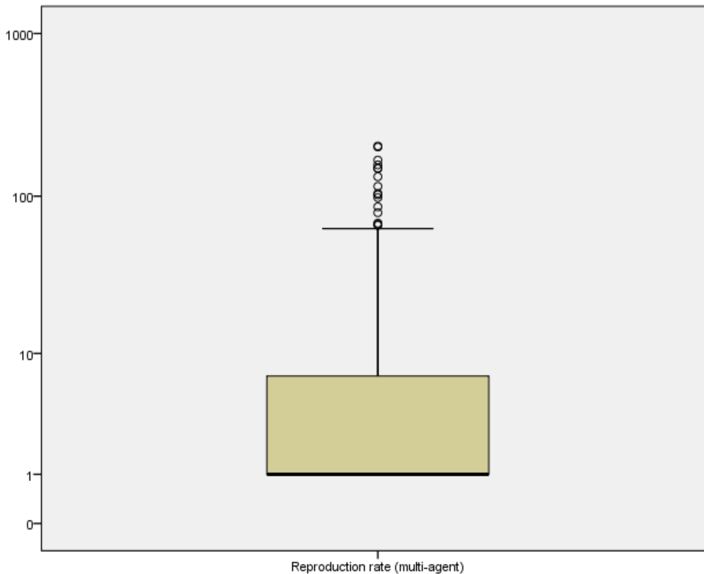


Figure 3.5: The distribution of R for the evaluated multi-agent systems.

The issues with reproducing failures in multi-agent systems operating in external environments originate from the inherent problems that arise when testing concurrent software [37]. However, this does underline the need for an automated testing tool that is able to run tests repeatedly at no additional costs. Improving the testability of multi-agent systems is an interesting direction for future work, for example by examining the applicability of solutions developed for mainstream programming languages (like Edelstein *et al.* [38]) to the agent-oriented paradigm.

3.6.5. Practical Use

An empirical investigation into the practical use of the testing framework was performed on a large set of solutions handed in by novice GOAL agent programmers, who were working in a total of 94 pairs. These pairs were given the same as-

signment[5], for which they had to develop a single agent and corresponding test(s) together. At three fixed points in time, they were asked to (voluntarily) send us their work up till that point; only the third (and final) version was actually graded. This set-up specifically allowed us to investigate failures in non-final submissions through a set of evaluations (i.e., using test conditions we formulated based on the assignment), as also suggested by Winikoff [8].

Tests

In Table 3.8, the descriptive statistics of the evaluations at each of the three hand-in moments are presented. The failures in each agent were determined by running them with a test constructed by ourselves (based on the assignment they were given).

Statistic	Mean	Std. Dev.	N
(1) No. of rules (1)	18.2	5.1	29
(1) P-failures	1.8	1.5	29
(1) G-failures	0.5	0.6	29
(1) A-failures	2.3	1.4	29
(1) No. of tests	0	0	29
(2) No. of rules	25.8	5.4	64
(2) P-failures	0.9	1.0	64
(2) G-failures	2.7	2.1	64
(2) A-failures	4.0	2.2	64
(2) No. of tests	2.8	10.8	64
(3) No. of rules	29.9	4.1	94
(3) P-failures	0.5	0.8	94
(3) G-failures	3.7	1.4	94
(3) A-failures	4.5	2.2	94
(3) No. of tests	30.8	19.2	94

Table 3.8: Descriptive statistics of the evaluations on the student assignments at each of the three hand-in moments.

At each hand-in moment, more students sent in their work (the final hand-in was the only mandatory one). As writing test condition was an explicit part at the end of the assignment, it is clear from the results that many students choose to do this at the end only. In addition, when the agent programs grow larger (i.e, in their number of rules) towards the final hand-in, the number of P-failures (i.e., failures in percept processing) simultaneously decreases each time whilst there is an increase in both A- and G-failures[6]. This is supported by correlation analysis at all hand-in moments ($p = .01$). Failures in goals also seem closely related to failures in actions, most likely because failures in goal management often cause problems in action selection. This is also supported by correlation analysis at all hand-in moments ($p = .01$). Finally, a correlation analysis of the number of failures (as determined by

[5]See https://github.com/eishub/BW4T.
[6]Note that missing functionality was not considered a failure in these evaluations.

us) with respect to the number of test conditions written by the students themselves showed significant results only at the final hand-in moment ($p = .05$), indicating a similar decrease ($r \approx -.25$) in all three failure categories when the number of tests conditions increases, thus cautiously confirming the usefulness of automated tests for these agents.

Questionnaires

At each hand-in, the pairs of students were also requested to fill in a short questionnaire when uploading their program. In Table 3.9, the descriptive statistics of these questionnaires are given. The students were asked to report the total number of hours spent on the assignment at each hand-in, and at the final deadline the percentage of that time they spent on testing and how effective they found the agent testing framework (on a Likert scale of 0-3).

Question	Mean	Std. Dev.	N
Hours spent (1)	3.6	2.3	29
Hours spent (2)	5.4	2.2	64
Hours spent (3)	14.0	4.5	94
Time spent on testing	19%	16%	94
Effectiv. of testing	1.1	0.8	94

Table 3.9: Descriptive statistics of questionnaire answers at each of the three hand-ins.

Interestingly, the amount of time spent on testing is quite high (and also quite different per student pair), but the effectiveness of the testing framework is not rated that highly. A qualitative evaluation of the feedback that students could provide at the final hand-in indicated some problems. The most frequently occurring feedback was (i) tests can take a long time to complete, (ii) failures can be hard to reproduce, and (iii) a full integration of the test results in Eclipse is missing. Most of these problems are related to the environment in which the (single) agent operated, as it could only run at a certain maximum speed, and by default randomly generates the world in which the agent operates. This indicates that agent environments may also require changes specifically to facilitate reproducible and repeatable testing. Moreover, even though an integration of the testing in the source-level debugger exists, students would like to see more integration of the test results in Eclipse itself, i.e., similar to the more graphical overviews that are given by mainstream programming languages.

Finally, a correlation analysis of the number of hours spent on the assignment (and the percentage of time spent on testing) with the amount of failures in the different categories supports the earlier conclusion that initially most of the failures are in the P category, whilst in the end most of the failures are in the G and A categories ($p = 0.01$).

3.7. Conclusions and Future Work

In this chapter, we proposed and defined an *automated testing framework* for cognitive agent programs, facilitating automated failure detection and reducing debugging effort that is required from a developer. We argue that modules are a natural unit for testing, and associate test conditions with modules of an agent program. We also introduced a test language that is used to specify test templates for detecting failure types. These test templates refine a failure taxonomy introduced by Winikoff [8]. A test approach has also been specified that explains how to instantiate test templates and derive test conditions for specific failure types. The main steps of this approach are to (i) define success in terms of functional requirements, (ii) test cognitive state updating, and (iii) classify failures that concern actions and goals.

The test language proposed is minimal in the sense that only two temporal operators are provided. We showed by analysing different agent program samples that the language is nevertheless sufficient for detecting failures in these programs. In particular, we were able to reproduce and detect all failures that were manually identified by Winikoff [8] using our automated testing framework. Interestingly, in about 77% of failures found in this reproduction, the testing framework also pointed to the code location of the corresponding fault. We demonstrated that our approach is not biased towards a specific sample of agent programs by applying the framework to other sample programs, and in a different environment. We were able to adequately detect failures by means of the automated testing framework, i.e., all agents eventually met all functional requirements after fixing the detected failures. We also showed that for single agents, test results are always consistent. However, when running multiple agents, a high number of repetitions might be needed to reproduce the same failure in some cases, suggesting an important direction for future work into the testability of multi-agent systems.

A concrete implementation of the proposed automated testing framework has been performed for the GOAL agent programming language, and detailed in this chapter, serving as a prototype for evaluation and as an example for other agent programming languages.

Empirical evaluation of a large set of test files and according questionnaires handed in by novice GOAL agent programmers also lead to several interesting results, suggesting additional directions for future work. For instance, developers spent a considerable amount of time on testing, indicating the importance of proper support for this task. However, some problems were present in the current implementation, mostly related to the fact that an external environment was used, causing problems for both reproducibility and (fast) repetition.

Finally, the focus of our work has been on automatically detecting failures. Even though our results are encouraging in that fault localization was facilitated by the testing framework, more work is needed for locating faults that correspond with these failures. In particular, we found that faults related to actions that are performed but should not have been performed are difficult to locate. Tools that can explain why these actions were performed might be useful here [39].

References

[1] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, *Automating failure detection in cognitive agent programs,* Agent-Oriented Software Engineering **6**, 275 (2018).

[2] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, *Using automatic failure detection for cognitive agents in Eclipse (AAMAS 2016 DEMONSTRATION),* in *Engineering Multi-Agent Systems: 4th International Workshop, EMAS 2016, Singapore, Singapore, May 9-10, 2016, Revised, Selected, and Invited Papers*, edited by M. Baldoni, J. P. Müller, I. Nunes, and R. Zalila-Wenkstern (Springer International Publishing, 2016) pp. 59–80.

[3] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, *Automating failure detection in cognitive agent programs,* in *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '16 (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2016) pp. 1237–1246.

[4] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, *Using automatic failure detection for cognitive agents in Eclipse (demonstration),* in *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '16 (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2016) pp. 1507–1509.

[5] C. Parnin and A. Orso, *Are automated debugging techniques actually helping programmers?* in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11 (ACM, New York, NY, USA, 2011) pp. 199–209.

[6] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling, *What do we know about defect detection methods?* Software, IEEE **23**, 82 (2006).

[7] ISO/IEC/IEEE, *24765:2017-9 Systems and software engineering – Vocabulary,* https://www.iso.org/standard/71952.html (2017).

[8] M. Winikoff, *Novice programmers' faults and failures in GOAL programs,* in *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14 (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2014) pp. 301–308.

[9] D. Weyns, H. Van Dyke Parunak, F. Michel, T. Holvoet, and J. Ferber, *Environments for multiagent systems state-of-the-art and research challenges,* in *Environments for Multi-Agent Systems: First International Workshop, E4MAS 2004, New York, NY, July 19, 2004, Revised Selected Papers*, edited by D. Weyns, H. Van Dyke Parunak, and F. Michel (Springer Berlin Heidelberg, 2005) pp. 1–47.

[10] M. Winikoff, *BDI agent testability revisited,* Autonomous Agents and Multi-Agent Systems **31**, 1094 (2017).

3

[11] R. H. Bordini, L. Braubach, J. J. Gomez-Sanz, G. O. Hare, A. Pokahr, and A. Ricci, *A survey of programming languages and platforms for multi-agent systems,* Informatica **30**, 33 (2006).

[12] M. Dastani, *Programming multi-agent systems,* The Knowledge Engineering Review **30**, 394 (2015).

[13] J. Dix, K. V. Hindriks, B. Logan, and W. Wobcke, *Engineering multi-agent systems (Dagstuhl seminar 12342),* Dagstuhl Reports **2**, 74 (2012).

[14] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah, *Testing in multi-agent systems,* in *Agent-Oriented Software Engineering X*, Vol. 6038 (Springer Berlin Heidelberg, 2011) pp. 180–190.

[15] Z. Zhang, J. Thangarajah, and L. Padgham, *Model based testing for agent systems,* Software and Data Technologies **22**, 399 (2008).

[16] R. Bordini, M. Dastani, and M. Winikoff, *Current issues in multi-agent systems development,* in *Engineering Societies in the Agents World VII*, Lecture Notes in Computer Science, Vol. 4457 (Springer Berlin Heidelberg, 2007) pp. 38–61.

[17] G. Caire, M. Cossentino, and A. Negri, *Multi-agent systems implementation and testing,* in *Proceedings of the 4th From Agent Theory to Agent Implementation Symposium, AT2AI-4* (2004).

[18] Z. Houhamdi, *Multi-agent system testing: A survey,* International Journal of Advanced Computer Science and Applications **2**, 135 (2011).

[19] R. Collier, *Debugging agents in Agent Factory,* in *Programming Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 4411, edited by R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni (Springer Berlin Heidelberg, 2007) pp. 229–248.

[20] M. Moreno, J. Pavón, and A. Rosete, *Testing in agent oriented methodologies,* in *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, Lecture Notes in Computer Science, Vol. 5518, edited by S. Omatu, M. P. Rocha, J. Bravo, F. Fernández, E. Corchado, A. Bustillo, and J. M. Corchado (Springer Berlin Heidelberg, 2009) pp. 138–145.

[21] J. Gómez-Sanz, J. Botía, E. Serrano, and J. Pavón, *Testing and debugging of MAS interactions with INGENIAS,* in *Agent-Oriented Software Engineering IX* (2009) pp. 199–212.

[22] R. Coelho, U. Kulesza, A. von Staa, and C. Lucena, *Unit testing in multi-agent systems using mock agents and aspects,* in *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems - SELMAS '06* (ACM Press, New York, NY, USA, 2006) p. 83.

[23] M. A. Khamis and K. Nagi, *Designing multi-agent unit tests using systematic test design patterns-(extended version),* Engineering Applications of Artificial Intelligence **26**, 2128 (2013).

[24] A. Carrera, C. A. Iglesias, and M. Garijo, *Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development,* Information Systems Frontiers , 1 (2013).

[25] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller, *Model-based test oracle generation for automated unit testing of agent systems,* Software Engineering, IEEE Transactions on **39**, 1230 (2013).

[26] A. M. Tiryaki, S. Öztuna, O. Dikenelli, and R. C. Erdur, *SUnit: a unit testing framework for test driven development of multi-agent systems,* in *Proceedings of the 7th international conference on Agent-oriented software engineering VII* (2007) pp. 156–173.

[27] E. E. Ekinci, A. M. Tiryaki, v. Çetin, and O. Dikenelli, *Goal-oriented agent testing revisited,* in *Agent-Oriented Software Engineering IX*, Lecture Notes in Computer Science, Vol. 5386, edited by M. Luck and J. J. Gomez-Sanz (Springer Berlin Heidelberg, 2009) pp. 173–186.

[28] J. Sudeikat, L. Braubach, A. Pokahr, W. Lamersdorf, and W. Renz, *Validation of BDI agents,* in *Programming Multi-Agent Systems: 4th International Workshop, ProMAS 2006, Hakodate, Japan, May 9, 2006, Revised and Invited Papers*, edited by R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni (Springer Berlin Heidelberg, 2007) pp. 185–200.

[29] D. Hovemeyer and W. Pugh, *Finding bugs is easy,* SIGPLAN Notices **39**, 92 (2004).

[30] M. Johnson, C. Jonker, B. van Riemsdijk, P. J. Feltovich, and J. M. Bradshaw, *Joint activity testbed: Blocks world for teams (BW4T),* in *Engineering Societies in the Agents World X*, Lecture Notes in Computer Science, Vol. 5881, edited by H. Aldewereld, V. Dignum, and G. Picard (Springer Berlin Heidelberg, 2009) pp. 254–256.

[31] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, *SWI-Prolog,* Theory and Practice of Logic Programming **12**, 67 (2012).

[32] T. Bosse, C. M. Jonker, L. van der Meij, and J. Treur, *LEADSTO: A language and environment for analysis of dynamics by simulation,* in *Multiagent System Technologies*, Lecture Notes in Computer Science, Vol. 3550, edited by T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. N. Huhns (Springer Berlin Heidelberg, 2005) pp. 165–178.

[33] M. Dastani, J. Brandsema, A. Dubel, and J.-J. C. Meyer, *Debugging BDI-based multi-agent programs,* in *Programming Multi-Agent Systems: 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009.*

Revised Selected Papers, edited by L. Braubach, J.-P. Briot, and J. Thangarajah (Springer Berlin Heidelberg, 2010) pp. 151–169.

[34] K. V. Hindriks, *Programming rational agents in GOAL,* in *Multi-Agent Programming: Languages, Tools and Applications*, edited by A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini (Springer US, 2009) pp. 119–157.

[35] V. J. Koeman and K. V. Hindriks, *A fully integrated development environment for agent-oriented programming,* in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection*, LNCS, Vol. 9086, edited by Y. Demazeau, K. S. Decker, J. Bajo Pérez, and F. de la Prieta (Springer International Publishing, 2015) pp. 288–291.

[36] R. Collier and J. Howell, *Vacuum world,* https://github.com/eishub/vacuumworld (2010), accessed: 2017-04-30.

[37] W. Pugh and N. Ayewah, *Unit testing concurrent software,* in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07 (ACM, New York, NY, USA, 2007) pp. 513–516.

[38] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur, *Framework for testing multi-threaded java programs,* Concurrency and Computation: Practice and Experience **15**, 485 (2003).

[39] K. V. Hindriks, *Debugging is explaining,* in *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 7455, edited by I. Rahwan, W. Wobcke, S. Sen, and T. Sugawara (Springer Berlin Heidelberg, 2012) pp. 31–45.

4

Facilitating Omniscient Debugging for Cognitive Agents

For real-time programs reproducing a bug by rerunning the system is likely to fail, making fault localization a time-consuming process. Omniscient debugging is a technique that stores each run in such a way that it supports going backwards in time. However, the overhead of existing omniscient debugging implementations for languages like Java is so large that it cannot be effectively used in practice.

In this chapter, we show that for agent-oriented programming practical omniscient debugging is possible. We design a tracing mechanism for efficiently storing and exploring agent program runs. We are the first to demonstrate that this mechanism does not affect program runs by empirically establishing that the same tests succeed or fail. Usability is supported by a trace visualization method aimed at more effectively locating faults in agent programs.

This chapter has been published in the International Joint Conference on Artificial Intelligence (2017) [1, 2].

4.1. Introduction

For traditional (cyclic) debugging to work, the program under investigation has to be deterministic. Otherwise, reproducing a bug by rerunning the program is likely to fail as it will not hit the same bug again or even hit different bugs [3]. Real-time programs like multi-agent systems are typically not deterministic. Running the same agent system again more often than not results in a different program run or trace, which complicates the iterative process of debugging. Chapter 3 also showed that the most frequently occurring type of failure in agent programs (a failure to select the right action) is caused by faults that occurred in a past state far from the point of detection. In other words, the root cause of a failure in an agent program is more often than not both far removed in time and in code (location).

Omniscient debugging is an approach to tackle these issues. Also known as reverse or back-in-time debugging, omnisicient debugging is a technique that originates in the context of object-oriented programming (OOP), allowing a programmer to explore arbitrary moments in a program's run by recording the execution. Such a 'time travelling debugger' is regarded as one of the most powerful debugging tools [4, 5].

However, omniscient debugging is still not widely adopted. An important reason for this is that existing (OOP) implementations have a significant performance impact, with slowdown factors ranging from 2 to 300 times. Moreover, most existing solutions are heavy on memory or disk space requirements, requiring tens of gigabytes for a single trace.

The fact that the *agent-oriented programming* (AOP) paradigm is based on a higher level of abstraction compared to most other programming languages provides an opportunity to apply omniscient debugging techniques with a significantly lower overhead. The premise here is that tracing for AOP can be based on capturing only high-level decision making events instead of the lower-level computational events of OOP. Tracing techniques for AOP would thus need tracing of significantly fewer events while still being able to reconstruct all program states, making omniscient debugging for AOP more feasible in practice than for e.g. OOP.

The main contribution of this chapter is the design of a tracing mechanism for cognitive agent programs that: (i) has a small impact on runtime performance; we show that our technique only has a 10% overhead instead of the much larger factors known from the literature (see Table 4.1); (ii) has virtually no impact on program behaviour; we empirically establish that the same tests succeed and fail with or without our tracing mechanism; (iii) can be effectively used for debugging, we propose a visualization technique tailored to cognitive agents and illustrate its application for fault localization. The key question we thus address is whether it is feasible and practical to apply omniscient debugging techniques to AOP without affecting testability.

4.2. Related Work

Omniscient debugging is based on the idea that a developer explores a run that failed by reversing back into its execution to locate the corresponding fault, rather

Work	Method	Storage req.	Slowdown	Storag loc.
[6]	Java bytecode instr.	100 MB/s	7-300x	in-memory[1]
[7]	Smalltalk bytecode instr.	1-100 MB/s	6-248x	in-memory
[8]	Java bytecode instr.	15 MB/s	10-115x	files
[9]	VM modification	300 MB/s	2-7x	in-memory
[10]	Java bytecode instr.	1-10 MB/s	2-15x	files
This work	Recording events	0.1 MB/s	1.1x	files

Table 4.1: A comparison of omniscient debugger implementations. Reported numbers have been rounded.

than trying to reproduce an observed failure in a separate (re)run as traditional (cyclic) debugging requires [3]. This is especially useful for programs that have non-deterministic aspects (e.g., randomness) and/or rely on external resources (e.g., agent environments), as such programs generally do not behave exactly the same way on each run.

Fundamentally, it is impossible to reverse the execution of a program because for many operations there is no way to take the state after the operation and infer the state before the operation [3]. *Omniscient debugging* facilitates 'going back in time' by recording an entire run of a program in a log, also called a *trace* of the run. Such a trace should allow any state of a program's execution to be correctly reconstructed. An intermediate form between cyclic and omniscient debugging is *record-replay debugging*, in which only the aspects of a run that cannot be re-constructed (i.e., by re-running) are recorded. With this method, going back to a previous point in the execution requires restarting the run, and then feeding the recorded aspects back in at exactly the right times. Record-replay debugging is easier to implement, but also more time-consuming for developers, as a complete restart and re-run is needed to go back only one state in an execution. Moreover, it is not always possible to record all required (non-deterministic) aspects of a program, especially when a program relies on external resources such as external environments. Finally, we note that tracing a program for debugging purposes is different from manual instrumentation like in Lam and Barber [11] or collecting (performance) measurements like in Helsinger *et al.* [12].

Although omniscient debugging has been a research topic since the 1970s, one of the first influential attempts to apply this debugging technique to a modern programming language (Java) was performed by Lewis and Ducasse [6]. In this chapter, a proof-of-concept omniscient debugger is presented with the intention of demonstrating an upper bound for the costs of collection and display. Every change to every accessible object or local variable is recorded in memory separately for each thread by adding instrumentation code before every assignment and around every method call. The author claims that this proof of concept is effective for many kinds of bugs. A similar effort for Smalltalk was performed by Hofer *et al.* [7], which also provides support for searching traces by queries (boolean expressions) specified in the language itself.

[1] There is a cut-off after 10.000 events on 32-bit systems.

The work of Pothier *et al.* [8] builds upon the work of Lewis, focusing on efficiency and usability. Events that are generated by (Java) bytecode instrumentation are stored in an on-disk database rather than in the program's memory space. Although this increases the capture cost, this provides better scalability as usually more disk space than memory is available, and reduces interference with the program memory itself. Moreover, this allows for post-mortem debugging by using previously recorded files. According to the authors, the benefits of omniscient debugging in quickly pinpointing hard-to-find bugs far outweigh any performance impact. The work of Lienhard *et al.* [9] aims to further address performance issues by tracing at the virtual machine level.

A related tool created by Ko and Myers [10] is *WHYLINE*, which allows developers to pose "why did" or "why didn't" questions about the output of Java programs. A trace is generated in memory through bytecode instrumentation, containing everything necessary for reproducing a specific execution. From this trace, a set of questions and according answers is generated. The authors note that their approach is not suited for executions that span more than a few minutes or executions that process or produce substantial amounts of data. However, their results do show that the approach enables developers to debug failures substantially faster.

Key aspects of the omniscient debuggers discussed in this section are compared with our mechanism in Table 4.1. We note that it is not possible to precisely compare the storage requirements (per second) and slowdown factors, as each work uses different programs (with varying amounts of activity in a run) in their evaluations, but the reported numbers do give a general indication of the various performance impacts.

A review of current state-of-the-art agent programming platforms shows that only three support something similar to a tracing mechanism[2]. 2APL [13] provides an event-based mechanism that captures so-called reasoning steps in-memory. Jason [14] provides a similar mechanism, but, as far as we can tell, only captures (snapshots of) the full state of an agent after each of its decision cycles. AFAPL [15] also captures the full state of an agent after each decision cycle. It is not clear if these tracing mechanisms provide sufficient support for implementing an omniscient debugging technique. 2APL and Jason's mechanisms do not scale well as they show fast growing memory usage, and, as a consequence, will quickly cause a significant impact on an agent's execution. The mechanism store the snapshots in files, but it is unclear what the associated performance impact is. From the three platforms discussed only AFAPL supports searching in a trace for the occurrence of specific beliefs, but none of the platforms support more advanced navigation, querying or filtering of a trace. None of these platforms is able to relate agent states that are stored to the agent program's source code, which is another feature that a developer needs for effectively locating faults.

Tracing	Cycles 1	Cycles 2	Cycles 3	Cycles 4	Total C.	Space 1	Space 2	Space 3	Space 4	Total S.
None	496	558	1453	749	3256					
Events	506	548	1432	780	3266	2MB	2MB	2MB	2MB	8MB
	+2%	-2%	-1%	+4%	+0%					
Changes	480	517	1281	696	2974	4MB	4MB	4MB	4MB	16MB
	-5%	-6%	-11%	-11%	-9%	+100%	+100%	+100%	+100%	+100%
Changes	469	501	1293	674	2937	5MB	5MB	6MB	5MB	21MB
+ sources	-2%	-3%	+1%	-3%	-1%	+25%	+25%	+50%	+25%	+31%
Full	77	78	83	79	316	394MB	385MB	391MB	390MB	1560MB
	-84%	-84%	-94%	-88%	-89%	+7780%	+7600%	+6417%	+7700%	+7329%

Table 4.2: An evaluation of different tracing methods by comparing the average amount of cycles over ten 1-minute runs of a system with 4 agents operating in the UT3 environment and the corresponding average amount of storage space that is required. The percentages that are given for a method are relative to the method directly above in the table.

4

4.3. Agent Trace Design

A tracing mechanism for cognitive agent programs should facilitate *reverting* an agent to any previous state by recording its execution. However, there are many ways to record a program's execution. Different solutions provide support for different techniques, ranging from record-replay debugging to support for full inspection which requires storing a *full trace*, i.e., all events, states, and actions performed in the run. Such a *full trace* that captures each individual state completely in practice is not feasible, as it takes 5-20x more time to execute an agent system and requires more than 50-200x the storage space needed for other mechanisms (see row *Full* in Table 4.2).

In this section, we take an incremental approach to the design of a tracing mechanism. We begin with an initial mechanism that stores a trace that captures as little information as possible but still provides sufficient information for record-replay. In this first step, a minimal trace is constructed based on *events* that, however, limits the options for a developer to (rapidly) locate points of interest in a trace and establish meaningful relations between these points. Step two extends the trace with information about *state changes* that allows efficient reconstruction of a previous state. In the third step, we add *source code information* associated with points in a trace to enable a debugger to more effectively explore the trace. At each step, we try to minimize the additional time and space resources needed and evaluate the impact of the tracing mechanism on the agent system's (runtime) performance. All evaluations were performed on a Linux server with a quad-core Intel i7 processor and 6GB of RAM. Finally, we discuss how to store traces for later use. In this chapter, we will show that the behaviour of different sets of agents in different environments is not significantly affected.

4.3.1. Tracing Events (Record-Replay)

In order to facilitate record-replay debugging, we need to determine which aspects *must* be stored in order to reconstruct any previous state of an agent program's execution by replaying. Assuming for the moment that agents themselves are deterministic (we will relax this assumption later), events such as percepts from an environment or messages from other agents would be the only items that need to be recorded in the trace. This is true because re-running an agent program does not guarantee the same events to be produced, as the environment is external and asynchronous and because multiple agents generally run concurrently in separate threads without a strict scheduling mechanism. By re-running the agent program with an initially empty set of events and by feeding the right events to the agent program at the right time to 'imitate' the environment and/or other agents, the run can be reconstructed from this *event trace* and the agent can be replayed.

An event trace can be implemented by storing *a snapshot of all events* that happened after each change. However, two optimizations can be applied to agent systems. First, events typically need to be stored only once per agent cycle. Second,

[2]No work has been published on the 2APL and Jason mechanisms; conclusions were drawn from own observations.

only changes in consecutive snapshots need to be stored. It is more efficient to only store events that have been added and deleted compared to the last snapshot as the rate of environment change typically is slow compared to the execution time of a single agent cycle and on consecutive cycles e.g. only a few changes to percepts are received. Storing the changes to events (e.g., with a listener pattern) thus only requires a simple comparison check with previous snapshots, and the required storage is linear in terms of this.

As a method to determine the performance impact of the tracing mechanism, we compare the average amount of cycles that agents performed in one minute. We used a randomly selected program from a pool of GOAL [16] multi-agent systems with four (different) agents that control bots in the highly dynamic UT3 environment [17]. The system first was ran ten times without any tracing enabled, and then ten times with the record-replay mechanism. Although runs are different due to the dynamics of the environment, the run settings (e.g., the map, the number of computer opponents, etc.) were identical in each case. We note that if a GOAL agent receives the same events as in a previous cycle and performs no new action in the environment, it (but not the environment entity) 'sleeps' until a new event occurs; we therefore actually report the number of 'effective cycles' an agent performed in one minute.

The results of these runs are summarized in the rows labelled *None* and *Events* in Table 4.2; columns match with each of the four agents with an additional column for totals. The results indicate that there is no significant difference in cycle numbers when the event tracing mechanism is enabled or not. Less than 2MB of storage space is needed per agent per minute, with about 100 events generated on average per second. We also established that space requirements grow linearly over time, i.e., no more than 20MB is required when agents are executed for ten minutes.

An important usability metric is how long it takes to reconstruct a program state. In a record-replay mode, it is clear that stepping from the final to the initial state takes no (significant) time at all, as this simply means restarting the agent. Moving forward in time to a next state is also fast as the agent does not need to be restarted. However, going just a single step backwards in time, i.e., to a previous state compared to the current state, an agent will need to be re-started and almost re-run completely to reconstruct that state.

To obtain an indication of our usability metric, we first established that re-playing our example agent programs to obtain the final state starting from the initial state using the event trace takes about 2.5 seconds. Given this measurement, navigating to an arbitrary state in a run in order to inspect that state will take about 1.25 second on average. Users, moreover, will want to evaluate queries to identify states they need to inspect, and in our test cases this will take more than 2.5 seconds as a query will need to be evaluated on each state that is reconstructed as well. For an agent that has run for just one minute (even though in a highly dynamic environment), this means a waiting time of more than 4% relative to execution time, which in practice is quite high.

4.3.2. Tracing State Changes

We have assumed that the execution of agent programs is deterministic, but this assumption does not hold for multi-agent systems as, e.g., the scheduling of execution steps of agent programs is non-deterministic. Moreover, a single agent can contain non-deterministic choice points like selecting a random element from a list or evaluating rules in a random order that will cause a different trace to be generated even with identical input. It is generally not possible to account for all such points, especially if they are at the knowledge representation level (and thus not explicitly represented in the agent programming language). The substantial waiting times are thus not the only reason why a record-replay approach for agent systems will not be useful in practice for agent systems.

In order to reduce the amount of time a navigation step in the trace takes on average and to facilitate non-deterministic agents, we need to make sure that an agent's state can be reconstructed without requiring re-execution. As storing each state in full is infeasible (see Table 4.2), we propose a mechanism that in addition to the changes to events also records all changes to an agent's cognitive state. The idea is that by recording all event and state changes, a navigation step can be performed by reconstructing a state by applying all changes between the current state and that target state.

The changes that need to be recorded differ per programming language. For the GOAL language, each change to an agent's beliefs or goals needs to be stored (besides the event changes related to percept and messages). For other agent programming languages that include notions like plans for example, a new intention that is scheduled or a change that pushes a new plan on an intention also needs to be recorded.

Note that it is not sufficient to store the actions performed by an agent program. For example, the actions of inserting a belief that the agent already has or dropping a goal the agent does not have, do not change the agent's cognitive state. In order to be able to navigate back in time, we need to know how we can 'roll back' each action to reconstruct a previous state. For each action performed by an agent that can change the agent's state, therefore, the real change brought about by that action given the agent's current state needs to be computed and stored in the trace. In other words, while executing an agent program, the mechanism needs to store aggregations of items that have been added to and/or removed from a state.

It is also not sufficient to 'instrument' program code to record state changes. Although most state changes correspond to an action that is performed as part of an agent program, they do not always originate directly from program code. For example, accomplishing a goal results in removing that goal automatically from an agent's goal base in GOAL. This means that the tracing mechanism has to be integrated into the virtual machine or interpreter of an agent platform.

As before, we analysed the performance of the *state change* tracing mechanism discussed in this section. The main results are summarized in the row labelled *Changes* in Table 4.2. Compared to event traces, we now see that on average the number of cycles has decreased by almost 10 percent. Even though this is still much better than the overhead introduced for traditional languages (see Table

4.1), further evaluation is required, and so in our evaluation we will determine whether agent behaviour has been changed to a point where it affects debugging or not. The space requirements have doubled, but on average still less than 4MB per agent per minute is required. The gain we achieve by increasing space usage is that our metric of navigation speed has been much improved: fully reversing an agent's state (either from first to last state but now also the other way around) takes only 0.5 seconds on average, less than 1% of the original execution time, and a substantial speedup compared to record-replay. As the time needed to evaluate queries remains the same, the speed-up factor for search queries on a trace will be lower, but only slightly so, as the time needed for evaluating a query on a state compared to reconstructing a state is almost negligible.

4.3.3. Tracing Source Code Locations

The state change tracing mechanism supports efficient reconstruction of a program's run. It also facilitates debugging by enabling fast querying of traces to identify unexpected state changes. But it does not yet support fault localization, as it is hard to relate such state changes to program code; the information about the state change itself does not specify where in the agent program it was brought about. For effective fault location, ideally, a tracing mechanism is fully integrated into the existing development facilities of an agent programming platform such as, for example, single-step execution debugging and automated testing. The integration with automated testing in general is relatively straightforward as our tracing mechanism makes a program run available for exploration immediately when a test failure is detected. Test conditions (that fail), moreover, also provide useful clues for executing search queries or applying filters on a trace.

The integration of our tracing mechanism with a source-level debugger, however, is more complicated. Ideally, a developer is able to follow the same stepping flow s/he can create with a source-level debugger but now also in the reverse direction using the recorded trace. But even if we are given a 'current' source code location and are able to revert to a previous state (given a trace), it is not clear how to 'reverse step' through the source code because there are many paths through a program that can result in the same state. It is not clear whether it is possible to reconstruct a single path from local state change information only, and even less clear how to do that efficiently. Instead, we therefore propose to support 'reverse stepping' by adding *source code location markers*, i.e., the traversed execution events/breakpoints to an agent's trace. This means storing a trace that not only records events and state changes, but also the full path of source code locations that is traversed while executing an agent program. In order to save space, each code location in an agent program is encoded as an integer number and this number instead of the file-based code references are stored together with an inverse mapping to retrieve the locations from these numbers.

To evaluate the impact of also storing code locations, we used the source-level debugging framework proposed in Chapter 2 to implement such a tracing mechanism and recorded all possible breakpoints, i.e. code locations that are traversed when 'single-stepping' an agent in forward mode. The main results are summarized

in the row labelled *Changes + sources* in Table 4.2. They show that recording source code locations does not have a big impact on the average number of cycles. The additional space needed to store traces was about 25%. It is worth noting though that space needed still only grows linearly over time, e.g., after ten minutes, our traces grew to roughly 50MB per agent (<0.1 MB/s).

4.3.4. Trace Storage

In order to minimize the impact of the tracing mechanism, the information that needs to be stored can be written to an (in-memory) queue. Due to the possible size of this queue and the memory requirements of agents themselves, this queue will need to be flushed to some more permanent storage in a thread(pool) that is separated from the agent runtime, preferably with a lower priority. One of the most efficient ways to do this is by using memory-mapped files [18], as they facilitate the best I/O performance for large files by mapping between a file and memory space, enabling an application to modify the file by reading and writing directly to the memory. Using files also facilitates debugging a trace at a later point in time (i.e., loaded from the file) and interaction with tools external to the agent runtime itself.

4.4. Evaluation

The main purpose of a tracing mechanism is to support debugging. It therefore is important to establish that the tracing mechanism does not significantly change the behaviour of an agent program. For debugging purposes, it is most important that the failures that occur in the agent system executed without tracing also occur when runs are being traced. It is also important to establish that the reproduction of a failure while tracing a program will not take many more runs and thus more time.

In this section, we empirically investigate and compare the performance and reproduction of a failure in an agent (system) with and without our 'state change and source location' tracing mechanism enabled. Whilst in the previous section we used Unreal Tournament (with 4 agents) for evaluation, in this section we will use the Blocks-World-for-Teams (BW4T; Johnson *et al.* [19]) environment with varying numbers of agents. Of all EIS-compatible environments [20] available to us ready for testing, UT3 and BW4T are the most dynamic.

4.4.1. Method

The method we used is to first run a given test set many times in order to identify all failures in an agent system (c.f. Chapter 3). We then ran multiple sessions in which those same tests were repeated in order to determine how many repetitions are needed (on average per session) to reproduce all the failures that were initially found. We used a time-out parameter as agent systems that produce many failures may run indefinitely without making any progress (i.e., producing no new results). Failures that are due to a time-out are ignored because these cannot be consistently reproduced. Our method, illustrated in Figure 4.1, uses these parameters:

- B: number of initial test runs for each agent system.
- T: number of seconds after which a test is aborted.
- N: number of 'sessions' for each agent system.
- M: maximum number of test runs in a single session.

The aim is to empirically determine the reproduction factor R: the number of times a run needs to be repeated to detect all failures in an agent system. We use R to evaluate our tracing mechanism, but our experiments also contribute useful insights into the testability of agent systems.

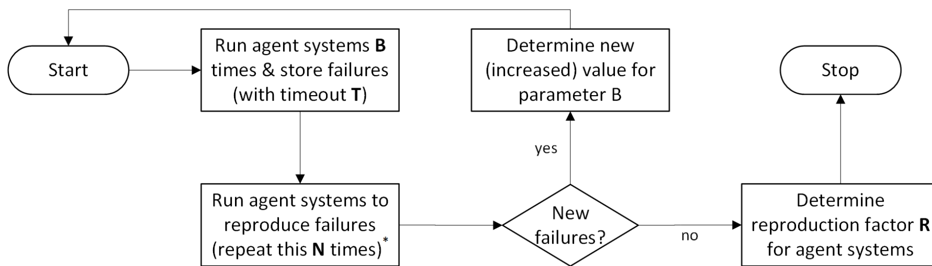


Figure 4.1: A flowchart of our evaluation method.

We used two sets of agent systems programmed in GOAL that control robots in the BW4T environment. They were created by pairs of first-year Computer Science bachelor students and handed-in with accompanying tests. The first set consists of 84 single-agent systems, and the second set of 42 multi-agent systems (3 agents).

4.4.2. Results

We applied our method to both sets of agents, with parameter B set to 100, T to 60 seconds, N to 10, and M to 1000. These parameters were chosen after an iterative process of running experiments to minimize the runtime whilst making sure that unreproducible nor new failures would be found in the repetition part of our method (see also the step to increase parameter B in Figure 4.1). We first ran agents with tracing turned off and then ran the same agents again with tracing turned on but skipped the first step (see Figure 4.1) as the goal is to establish whether failures are reproduced also when tracing is turned on. In total, for our final experiment, almost 23,000 runs were performed with a total runtime of about 330 hours.

We found a significantly lower number of failures for the single-agent systems (on average 0.3 failures, with a maximum of 8 failures) than for the multi-agent systems (on average 4.3 failures, with a maximum of 21 failures). In a rather static environment like BW4T with a low number of failures, we found that a single agent's failure set can be reproduced on every single run both with and without tracing enabled, i.e., $R = 1$. This is very different for multi-agent systems where on

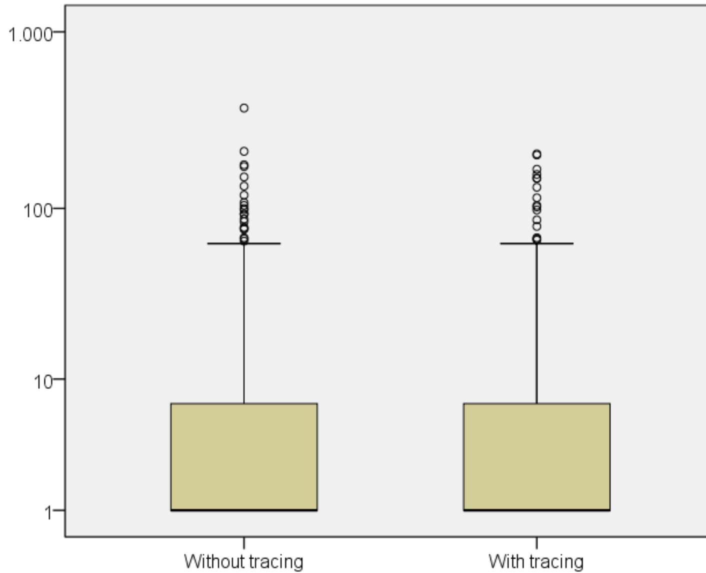


Figure 4.2: Distribution of R with and without tracing.

average $R = 11$ runs were needed to reproduce all failures that were initially found. Most importantly, this establishes that all failures could be reproduced (and no new failures were introduced) when the tracing mechanism is used. The distribution of R for both with and without tracing enabled is shown in Figure 4.2. Even though on average only 11 runs were needed, in a few cases up to even 300 repetitions were needed to reproduce all failures. The high number of runs required in these cases provides a strong indication that omniscient debugging is a technique that is needed in practice to be able to debug multi-agent systems. When comparing the distributions for R using a Wilcoxon signed-rank test, no statistically significant difference is found ($Z = -0.79$, $p = 0.43$). This provides additional support for the claim that the tracing mechanism does not impact the agent system's execution. Finally, we note that a few extreme outliers where >500 runs were required were excluded from these results.

4.5. Visualizing Traces

For efficient fault localization, it needs to be easy for a developer to identify states in a program's execution that are related to the failure under investigation. Moreover, a developer should not get lost in navigating between these states, but always have a sense what point in the execution s/he is evaluating and how the current state affected the execution.

We adapt the concept of a *space-time view* first developed in Azadmanesh and Hauswirth [21] in the context of Java programming to cognitive agent programming. A space-time view is a table that is structured along space and time dimensions,

where the rows in the table correspond to the space dimension, which is composed of the different elements in a state that are traced. Each cell indicates whether an element was modified by executing an operation or only accessed for inspection at a specific time (the columns in the table).

		2	3	4	5	6	7
nrOfPrintedL...		Modification			Inspection		
initCounter/0			Exit				
helloWorld1...				Entry			
print/1						Call	Action
updateCoun...							

		2	3	4	5	6	7
nrOfPrintedL...		Modification			Inspection		

Figure 4.3: Space-time view (top) and filtered version (bottom)

For cognitive agents, the elements in a space-time view that are traced are the agent's events, beliefs, goals, actions, plans, and/or modules (i.e., sets of decision or plan rules). Assuming a basic representation of a name with associated parameters is used to represent these elements, we use the corresponding *signatures* as the rows in the space dimension. For example, the signature `print/1` in Figure 4.3 represents a `print` action with one parameter. Each point (event, state change, source code location) in a trace represents a step (column) in the time dimension. Multiple space elements (signatures) can be used in a single step, e.g., evaluating a query may require accessing several beliefs and goals. The cells in our space-time view contain information about how an element was used at a particular step, which differs per type of element (e.g., a belief can be modified or inspected, an action or plan can be called and performed, a module can be entered or exited). Empty cells indicate the element was not used. An example of a space-time view for a simple agent is shown in Figure 4.3.

A developer can use and manipulate a space-time view in several ways. The signatures listed in the space time view can be ordered based on type (beliefs next to beliefs) or alphabetically (using the signature names). A user can also apply queries or filters to a trace both textually as well as through selecting cells of interest or rather cells that should be hidden in the table; see, for example, the bottom table where only the first row is selected by a user in Figure 4.3. A user can click on any cell in the table in order to step the agent to the state matching that cell's column (either forwards or backwards through its execution), allowing a developer to use all debugging tools (e.g., inspecting or modifying an agent's beliefs and goals) in that specific historic state.

We illustrate the use of such a space-time view for analysing a failure of the following example test condition associated with a BW4T agent program:

```
goal(holding(B)), bel(atBlock(B)) leadsto done(pickUp(B))
```

This condition expresses that if the agent has the goal to hold block `B`, and believes it is at the block, that it should (eventually) pick up `B`. A failure to do so will lead to failure of the test condition (i.e., when the agent is terminated). Without an omni-

scient debugger, a developer would need to restart the agent, navigate to a point where the goal-believe query holds (*assuming* it will at some point in the *restarted* run), and continue by manually stepping to try to understand why the action is not performed. With an omniscient debugger, we do *not* need to restart the agent, and can use the clues provided by the test condition itself to navigate to the last time that `holding/1` and `atBlock/1` were modified in the space-time view. We can do so either by double-clicking the corresponding cell, or, even faster, by using the query `goal(holding(B)), bel(atBlock(B))` to filter the trace. Note that such a point *must* exist in the run as the test condition failed on the exact same run that was traced. Because our tracing mechanism also traces source code locations and is integrated with a source-level debugger, a developer can now step from that point through the source code *as if* it is executed for the first time (and go backwards whenever needed). In our example, it quickly became clear to the developer that some decision rules were incorrectly ordered, which prevented the `pickUp` action from being executed.

4.6. Conclusions and Future Work

In this chapter, we proposed a tracing mechanism design that supports omniscient debugging for cognitive agents, a technique that facilitates debugging by moving backwards in time through a program's execution. Using a prototypical implementation of the tracing mechanism in the GOAL agent programming language, we evaluated and demonstrated empirically that the mechanism is efficient and does not substantially affect the runs of program in the sense that the same failures can be reproduced when the mechanism is turned on and off. This essentially shows that our mechanism is fast enough and can be used in practice for debugging failures without a need to rerun a program.

We also introduced a trace visualization method tailored to cognitives agents based on a space-time view of the execution history. A developer can navigate this view, evaluate queries on a trace, and apply filters to it to obtain views of only the relevant parts of a trace. Our approach is integrated with a source-level debugger and traces source code locations, which enables a developer to single-step through a program's execution history and facilitates fault localization.

Future work will include a user study to evaluate the usability of our omniscient debugging approach for programmers. Our findings that it can be hard to reproduce a failure at least sometimes also prompt the need for further investigation into how failure reproduction for multi-agent systems can be improved. Finally, we believe that our tracing mechanism can provide a starting point for a history-based explanation mechanism that can automatically answer questions such as 'why did this action (not) happen?' [22].

References

[1] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, *Omniscient debugging for cognitive agent programs,* in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17 (AAAI Press, 2017) pp. 265–272.

[2] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, *Omniscient debugging for GOAL agents in Eclipse,* in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17 (AAAI Press, 2017) pp. 5232–5234.

[3] J. Engblom, *A review of reverse debugging,* in *System, Software, SoC and Silicon Debug Conference (S4D), 2012* (2012) pp. 1–6.

[4] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009).

[5] G. Bracha, *Debug mode is the only mode,* https://gbracha.blogspot.nl/2012/11/debug-mode-is-only-mode.html (2012), accessed: 2017-02-19.

[6] B. Lewis and M. Ducasse, *Using events to debug Java programs backwards in time,* in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03 (ACM, New York, NY, USA, 2003) pp. 96–97.

[7] C. Hofer, M. Denker, and S. Ducasse, *Design and implementation of a backward-in-time debugger,* in *NODe 2006*, Vol. P-88 (GI, Erfurt, Germany, 2006) pp. 17–32.

[8] G. Pothier, r. Tanter, and J. Piquer, *Scalable omniscient debugging,* SIGPLAN Notices **42**, 535 (2007).

[9] A. Lienhard, T. Gîrba, and O. Nierstrasz, *Practical object-oriented back-in-time debugging,* in *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, edited by J. Vitek (Springer Berlin Heidelberg, 2008) pp. 592–615.

[10] A. J. Ko and B. A. Myers, *Extracting and answering why and why not questions about Java program output,* ACM Transactions on Software Engineering and Methodology **20**, 4:1 (2010).

[11] D. Lam and K. Barber, *Comprehending agent software,* in *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05 (ACM, New York, NY, USA, 2005) pp. 586–593.

[12] A. Helsinger, R. Lazarus, W. Wright, and J. Zinky, *Tools and techniques for performance measurement of large distributed multiagent systems,* in *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03 (ACM, New York, NY, USA, 2003) pp. 843–850.

[13] M. Dastani, *2APL: a practical agent programming language,* Autonomous Agents and Multi-Agent Systems **16**, 214 (2008).

[14] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak using Jason* (John Wiley & Sons, Ltd, 2007).

[15] R. Collier, *Debugging agents in Agent Factory,* in *Programming Multi-Agent Systems: 4th International Workshop, ProMAS 2006, Hakodate, Japan, May 9, 2006, Revised and Invited Papers*, edited by R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni (Springer Berlin Heidelberg, 2007) pp. 229–248.

[16] K. V. Hindriks, *Programming rational agents in GOAL,* in *Multi-Agent Programming: Languages, Tools and Applications*, edited by A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini (Springer US, 2009) pp. 119–157.

[17] K. V. Hindriks, B. van Riemsdijk, T. Behrens, R. Korstanje, N. Kraayenbrink, W. Pasman, and L. de Rijk, *Unreal GOAL bots,* in *Agents for Games and Simulations II: Trends in Techniques, Concepts and Design*, edited by F. Dignum (Springer Berlin Heidelberg, 2011) pp. 1–18.

[18] D. S. Roselli, J. R. Lorch, T. E. Anderson, *et al.*, *A comparison of file system workloads,* in *USENIX annual technical conference, general track* (2000) pp. 41–54.

[19] M. Johnson, C. Jonker, B. van Riemsdijk, P. J. Feltovich, and J. M. Bradshaw, *Joint activity testbed: Blocks World for Teams (BW4T),* in *Engineering Societies in the Agents World X: 10th International Workshop, ESAW 2009, Utrecht, The Netherlands, November 18-20, 2009. Proceedings*, edited by H. Aldewereld, V. Dignum, and G. Picard (Springer Berlin Heidelberg, 2009) pp. 254–256.

[20] T. M. Behrens, K. V. Hindriks, and J. Dix, *Towards an environment interface standard for agent platforms,* Annals of Mathematics and Artificial Intelligence **61**, 261 (2011).

[21] M. R. Azadmanesh and M. Hauswirth, *Space-time views for back-in-time debugging*, Tech. Rep. 2015/02 (University of Lugano, 2015).

[22] K. V. Hindriks, *Debugging is explaining,* in *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 7455, edited by I. Rahwan, W. Wobcke, S. Sen, and T. Sugawara (Springer Berlin Heidelberg, 2012) pp. 31–45.

5

Designing a Cognitive Connector for Complex Environments

One issue that needs to be addressed time and again is how to create a connector for interfacing cognitive agents with richer environments. Cognitive agents use knowledge technologies for representing state, their actions and percepts, and for deciding what to do next. Issues such as choosing the right level of abstraction for percepts and action synchronization make it a challenge to design a cognitive agent connector for more complex environments.

The leading principle for our design approach to connectors for cognitive agents is that each unit that can be controlled in an environment is mapped onto a single agent. We design a connector for the real-time strategy (RTS) game StarCraft and use it a case study for establishing a design method for developing connectors for environments. StarCraft is particularly suitable to this end, as AI for an RTS game such as StarCraft requires the design of complicated strategies for coordinating hundreds of units that need to solve a range of challenges including handling both short-term as well as long-term goals. We draw several lessons from how our design evolved and from the use of our connector by over 500 students in two years. Our connector is the first implementation that provides full access for cognitive agents to StarCraft: Brood War.

This chapter is to be published in the book Engineering Multi-Agent Systems (2018), an extension of work published in the International Workshop on Engineering Multi-Agent Systems (2018) [1], in turn an extension of work published in the Conference on Autonomous Agents and Multi-Agent Systems (2018) [2].

5.1. Introduction

Multi-agent systems, consisting of multiple autonomous agents interacting with an external environment, have been promoted as the approach for handling problems that require multiple problem solving methods, multiple perspectives, and/or multiple problem solving entities [3]. In the past twenty years, the research community has combined multi-agent system (MAS) concepts and approaches into mature frameworks for agent-oriented programming (AOP) [4, 5]. Current cognitive agent technology thus offers a viable and promising alternative to other approaches for engineering complex distributed systems [6, 7]. However, Hindriks [6] also concludes that "if [cognitive] agents are advocated as the next generation model for engineering complex, distributed systems, we should be able to demonstrate the added value of [multi] agent systems." Designing a connector that can demonstrate this added value by connecting cognitive agents with an environment that puts strict real-time constraints on the responsiveness of agents, requires coordination at different levels (ranging from a few agents to large groups of agents), and requires complex reasoning about long-term goals under a high level of uncertainty is not a trivial task. The connectors that are currently available for use with cognitive agent systems have remained rather simple, and thus do not fully demonstrate the added value of cognitive agent technology.

In this chapter, we aim to establish a *design approach for developing connectors for complex environments*, aimed at facilitating the development of more connectors that can be used to demonstrate the ease of use of cognitive technologies for engineering large-scale complex distributed systems for challenging environments. We believe that RTS games that deploy large numbers of units provide an ideal case study to this end [8, 9]. The basic idea is to control each unit with a cognitive agent. Based on this, and in accordance with Google (DeepMind) and many other AI researchers [10, 11], we believe that StarCraft is the most suitable RTS game to target in our case study. Moreover, several popular competitions exist for StarCraft AI that can serve as a benchmark for implementations that use cognitive technologies [11]. By carefully designing and efficiently implementing a cognitive agent connector to StarCraft, and then testing this connector with large groups of students, we iteratively refine our approach for the development of cognitive agent-environment connectors.

Our focus in this work is on the case study of designing a connector that enables and facilitates the use of cognitive agent technology for engineering strategies for StarCraft (Brood War) based on a *one-to-one unit-agent mapping*, which is different from most existing StarCraft AI implementations. This unit-agent mapping introduces important challenges that need to be addressed:

1. The connector should facilitate a MAS that operates at a level of *abstraction* that is appropriate to cognitive agents.

2. The connector should be sufficiently *performant* in order to support a sufficient variety of viable MAS implementations using cognitive agents (i.e., both different approaches to implementing strategies as well as the use of different agent platforms).

In other words, the connector design should not force a cognitive MAS to operate at the same level of detail as bots written for StarCraft in C++ or Java, but also not promote the other extreme and abstract too much (e.g., clearly the extreme abstraction of providing a single action 'win' is not useful). To make optimal use of the reasoning typically employed by cognitive agents, the connector should leave low-level details to other control layers whilst still allowing agents sufficiently fine grained control.

5.2. Related Work

Connectors that support connecting cognitive agent technology to games have been made available for other games [12]. So far, however, most connectors have remained rather simple. The most complex cognitive multi-agent connectors that have been made available so far, are connectors for Unreal Tournament [13]. The design of such a connector involves similar issues related to the facilitated level of abstraction and the resulting performance as in this work. However, the resulting implementation as reported on by Hindriks *et al.* [13] does not support running more than 10 agents, whereas for a StarCraft interface we need to connect hundreds of cognitive agents to control the hundreds of units in game. Moreover, corresponding agent systems for Unreal Tournament generally offer only a very restricted set of actions that agents can perform (i.e., mostly just a "go to" action because other middleware software is used to take care of path planning, shooting, etc.) or communication (i.e., mostly just informing others about enemy positions), limiting the complexity of decision making that is required. Relatively speaking, compared to StarCraft, the diversity in strategies or tactics that can be deployed is rather small. Another problem related to Unreal Tournament is that games cannot be sped up, complicating testing and debugging. It is therefore not feasible to derive a design approach for connectors to richer environments from this work.

RTS games are widely regarded as an ideal testbed for AI [9, 10]. An RTS game like StarCraft involves long-term high-level planning and decision making, but also short term control and decision-making with individual units. This distinction between respectively strategical and tactical decision making is generally referred to as *macro* and *micro* respectively. These factors and their real-time constraints with hidden information make RTS games like StarCraft ideal for iterative advancement in addressing fundamental AI challenges [9]. Although machine learning solutions have been applied to some problems at the micro level, learning techniques have not been successfully applied to other aspects, mainly due to the vast state spaces involved [11]. The concepts of cognitive agents seem to be a good fit for addressing these challenges, allowing individual cognitive agents to reason about their tactical decision making whilst also inherently facilitating communication to make decisions at a joint strategical level. The reasoning typically applied by cognitive agents seems to lend itself for macro really well, but such systems can potentially employ learning techniques to perform specific sub-tasks (at the micro level) as well. A cognitive agent connector can also facilitate the use of MAS as an approach for allowing several individual AI techniques to work together.

The work of Weber *et al.* [14] recognizes the value of agent-oriented techniques

5

for StarCraft AI. Their "EISBot" uses a reactive planner combined with external components like case-based reasoning and machine learning. Similar to multi-agent systems, the concepts of percepts and actions are used. However, there is only a single 'agent' that is compartmentalized into several specific managers. This approach is thus still based on a single-bot approach, whilst in this work, we instead aim to design a connector for multi-agent systems in which each in-game unit is connected to an individual cognitive agent. Moreover, it is not made clear which percepts and actions are provided, and what the gain in terms of abstraction level and the loss in terms of performance in this implementation is, as the focus is on the implementation of the StarCraft bot itself, instead of on the design of a (generic) connector as in this paper.

The prototypical RTS game is StarCraft [11], originally developed by Blizzard in 1998, but still immensely popular both in (professional) gaming and AI research. An API for StarCraft (Brood War) has been developed for several years: BWAPI [15]. BWAPI reveals the visible parts of the game state to AI implementations, facilitating the development of competitive (non-cheating) bots. Several dozens of such bots have been created with this API, mostly written in C++ or Java, aimed at participating in one of the tournaments that are being held for StarCraft AI implementations. However, this work does not directly facilitate cognitive agents that use knowledge technologies and realise a one-to-one unit-agent mapping.

A first attempt at creating a cognitive interface for StarCraft was performed by Jensen *et al.* [16]. In this work, a working proof-of-concept that ties in-game units to cognitive agents was introduced. However, it does not address the major challenges such an implementation faces concerning the level of abstraction and corresponding performance, as we do in this work. When using this connector, it is not possible to create viable (diversities of) strategies, as the range of strategies it supports is quite limited. This connector only offers a small subset of all possible actions associated with each unit in the game, and the percepts made available by the connector do not provide sufficient information for in game decision making either. In this work, we aim to allow virtually any strategy to be implemented with a sufficient level of performance using a cognitive agent connector based on the design approach we propose.

5.3. Case Study: StarCraft

In StarCraft, each of the three playable races have their own set of unit types, with roughly 15 types of air/ground units and 15 types of buildings per race. Although many races share similar types of buildings (e.g., depots to bring resources to), there are also substantial differences to take into account (e.g., one race requiring units to 'morph' into a different type of unit). For most types of units, there are usually multiple 'instances' (i.e., individual units) in a game, thus allowing anywhere from 5 up to 400 units representing one army in the game at a certain time. Depending on factors such as game length, the average number of units for an army in a typical game at any point in time is around 100, although many units will also die during the game (i.e., the total number of agents used is much higher). Performance is thus of vital importance, as a substantial performance impact caused by

large amounts of percepts for example, will limit the amount of viable strategies.

Our cognitive agent connector to StarCraft was developed and refined in three iterations. We draw several general lessons from these iterations, which we have incorporated into our proposal for a connector design approach. Initially, a pilot was held with around 100 Computer Science master's students that worked in groups on creating a StarCraft bot using this connector. Shortly after, over 200 first year Computer Science bachelor's students did the same with an improved version of the connector, being the largest StarCraft AI project so far. We continued development of the connector after this project, and made several additional improvements, after which 300 first year Computer Science bachelor's students used the 'final version' of our connector.

5.4. Connector Design Approach

In this section, we discuss our design approach for a cognitive agent connector. The core of such a connector consists of three components: (i) the *entities* that are provided for agents to connect to (i.e., units in an RTS game), (ii) the outputs that are generated by each entity (and thus which *percepts* a corresponding agent receives), and (iii) the inputs that are available for each entity (and thus which *actions* an agent controlling the entity can perform). This structure is illustrated in Figure 5.1. Each of these aspects will be discussed, starting with general guidelines, their application to our case study of StarCraft, and the refinements that were made after practical use of the StarCraft connector. Next, key steps for evaluating whether the connector design is fit for use in practice for developing cognitive MAS will be given and performed for our connector.

We make some basic assumptions about the architecture of a *cognitive agent*, as illustrated in Figure 5.2. We assume such an agent pro-actively reasons about the *actions* that it should take based on (for example) its goals and beliefs in some fixed *decision cycle* that is asynchronous from the environment in which it operates (for a certain *entity* in that environment), from which it receives information through *percepts*. Multiple agents can work together in one multi-agent system, which is not centrally controlled but does facilitate direct *messaging* between (groups of) agents. Our connector makes use of the Environment Interface Standard [17] in order to facilitate interacting with MAS platforms.

5.4.1. Micro and Macro Management

In complex environments such as StarCraft, a crucial distinction exists between top-down strategical decision making (macro) and bottom-up tactical decision making (micro). The basic assumption that we make is that a connector needs to provide support for a multi-agent approach based on a one-to-one unit-agent mapping, which inherently facilitates decision making from a bottom-up perspective. At the micro level, *every unit that is active in the environment should be mapped onto an entity* that a cognitive agent can connect to in order to control the behaviour of the unit. For StarCraft, this thus means that any moving or otherwise active unit such as a building will be controlled by a cognitive agent.

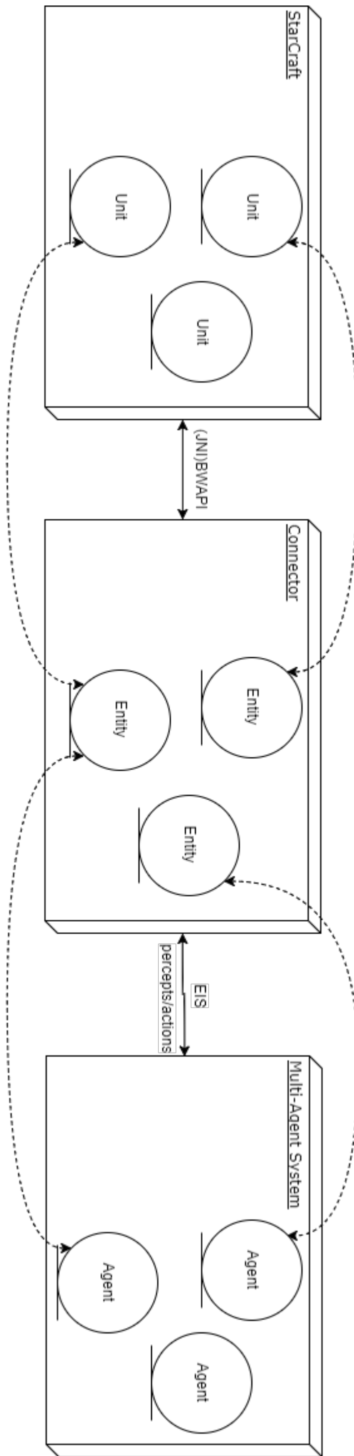


Figure 5.1: An overview of the various components, with StarCraft on the left, our connector in the center, and a cognitive agent system playing the game on the right.

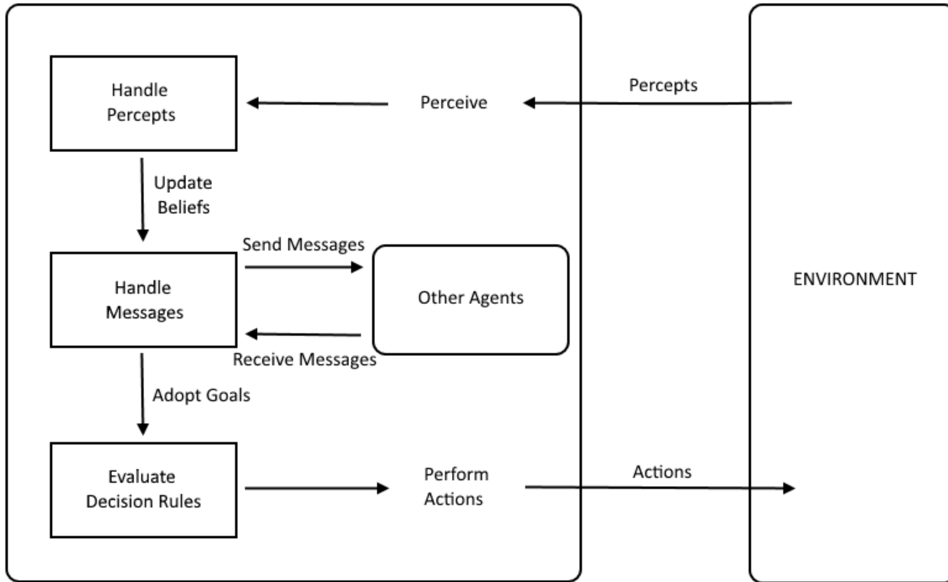


Figure 5.2: The assumed structure of a cognitive agent in a multi-agent system (left) interacting with an external environment (right).

Although we initially assumed that the emergent behaviour from these agents would be sufficient to cover the strategical aspects, in practice this was hindered by the high dynamicity of an environment such as StarCraft, for example illustrated by the fact that any unit can be killed at any point in time. To facilitate macro management, we therefore have introduced a new, special kind of entities, so-called *managers*, which are made available by the connector. Managers do not match with unique in-game units, and as such they do not naturally have percepts or actions associated with them. However, as they still need to be informed about the state of the game in order to perform strategical decision making, they instead should have the ability to receive desired global information through percepts, as for example indicated by a developer in the initialization settings of a MAS.

Manager agents are especially useful to reason about groups of units. For example, without managing agents, all agents for resource gathering units in StarCraft (of which there are generally several dozen) would have to process information about the available resources and resource depots (i.e., subscribe to the relevant percepts and handle them), and then coordinate amongst each other about the division of tasks (i.e., implement some decentralized messaging protocol). Instead, a single manager agent can be the only one to have to deal with all the information about resources, and then use this information to assign a task to each resource gathering unit (i.e., through messaging), whilst in contrast the agents for those units would still handle defending themselves for example. This significantly reduces the total amount of percept processing and message sending that is required

in such a situation. Moreover, in our case study we found that there is a need for dynamically adding or removing managers in order to for example temporarily centralize the reasoning for a group of attacking units, which is another frequently occurring situation in which using managers is beneficial for both performance and the effectiveness of the coordination between the relevant agents. The specific type and choice of managers that are made available by a connector and the resulting organizational structure is, however, not specified in our design approach so as to facilitate as many multi-agent system structures as possible. As there is information that is specific to certain units (and thus specific agents), and each unit has its own set of actions (which a single agent needs to call), it is not possible to completely centralize the reasoning.

Because our approach is to provide an entity (i.e., to which an agent can connect) for each unit, and the available actions for each unit are mainly defined by the (interface to) the environment itself, the main challenge when balancing the level of abstraction with the resulting performance is in determining the percepts that are available. As we assume cognitive agents here that explicitly represent their beliefs and goals, this essentially means we need to design an ontology that includes all relevant concepts for representing and reasoning about the environment at an appropriate level of abstraction.

5.4.2. Local and Global Information

The set of available percepts determines what information a specific entity 'sees' during the game, and thus what information its corresponding agent will receive. Percepts have a *name* to describe them and a set of *arguments* that contain the actual data. For example, a percept could be defined as `map(Width, Height)`, and an agent could then receive `map(96, 128)` in a match. In order to determine the percepts that are created for each type of unit, our approach proposes several design guidelines. A key foundation of our approach to handling information from complex environments such as StarCraft is that there is a difference between '*local*' information that is specific to a certain unit in the game (e.g., a unit's health) and '*global*' information that is potentially relevant to all units (e.g., the locations of enemy units). An agent should be able to *perceive all local information* that is specific to its corresponding unit's state, whilst a manager agent should be able to *perceive all global information* that is needed for its strategic (macro) reasoning. However, pieces of global information might also be needed in the agent for a specific unit (e.g., nearby enemy units in StarCraft).

To this end, we initially pushed all global information to all unit and manager entities, as a connector cannot determine which parts of this information a specific agent will need. However, our case study showed that this caused a significant performance impact with larger numbers of units. We have therefore found it useful to provide specific mechanisms to a developer to fine-tune the delivery of global percepts. Through the connector's initialization settings, a list of desired 'global information' (i.e., names of percepts) can be given ("subscribed to") for each unit type. For example, the (pseudocode) initialization rule `zergHatchery: [friendly, enemy]` will ensure that all agents for all *Zerg Hatchery* entities in a match will

receive information about all friendly units and all visible enemy units. In this way, a developer can decide which information is relevant for certain agents, instead of such information being sent to agents at all times. This mechanism can also be used for specifying in more detail which global information a certain manager agent needs to be made aware of. Finally, we assume that when local information is needed for macro reasoning, this can be sent to the appropriate manager agent by the agent for a specific unit within the agent platform; it is thus not required to handle this within the connector (design) itself, as illustrated by the wave-shape in Figure 5.3.

Micro
Macro
Local
Global

Figure 5.3: Main design approach for organizing information into local and global percepts for micro (unit) or macro (manager) entities.

The ease of use of the percepts for an agent programmer should also be taken into consideration, i.e., by grouping related pieces of information together. The design guideline here is that one should *only group sets of parameters that naturally belong together*. Moreover, to avoid having to deal with different kinds of percepts for each type of unit, a design guideline is that *the percepts should be as generic as possible in order to facilitate re-use* between different agents. This guideline is aimed at reducing the number of different concepts introduced in our percept ontology, and thus aims for efficiency of design. An example of this is the `status` percept for each unit, as its structure (i.e., the set of parameters) is the same for each unit, even though not all information might be relevant for each unit (not all units use energy for example, but a unit's energy level is always provided in the percept). This also allows for specifying generic code for handling the `status` percept for all agents only once in the program, instead of having to specify this specifically (and nearly identically) for each unit type; special cases for certain types of units can then be programmed only where necessary.

Performance

One of the main challenges is how to deliver all percepts while guaranteeing sufficient performance levels. It is important to manage the percept load of individual agents, as creating the information needed for percepts (i.e., in the connector) and relaying that information to one or multiple agents who then have to make this information available for use in reasoning (i.e., by representing them in a Prolog base) is the most resource intensive task in a connector. In contrast to actions, of

which usually at most one is selected per decision cycle, there are usually many percepts (all containing various amounts of information) sent to each agent per decision cycle. We therefore introduce a number of optimization guidelines which aim to either reduce the total number of percepts an agent will have (to store) or the amount of updates to this set of percepts that an agent will have to process.

Complex environments have a lot of static information to which all individual agents may need to have access, like what a certain unit costs to produce or what kinds of units a certain building can produce in StarCraft. Because such environments also introduce many units (and thus many agents), the initialization costs for such information for each of these agents can have a rather big impact on a connector's performance. To avoid this issue as much as possible, we introduce another design guideline to *only create percepts for information that changes in a single match or between matches*. Static information is better suited to be encoded in the agent system itself instead of being sent through percepts, as this will significantly reduce the performance when initializing an agent (which as aforementioned can happen many times during a game as large numbers of units come and go almost constantly). To this end, information that is fixed by the game itself can be coded as a separate part of the ontology that can and needs to be loaded only once at the start of the game. Agents will still need to be informed about changes between matches, e.g., map-specific information should not be included in the 'fixed part' of the ontology. Another guideline to keep the number of percepts low is to ensure that *no data is sent through percepts that can either be calculated based on other data* (e.g., the number of friendly units by counting the number of percepts about their status), *or retrieved from other agents* (e.g., the position of a friendly unit). Relaying information (like friendly unit positions) through messaging between the agents in a MAS is usually much more efficient, as an agent programmer can then selectively choose at which times and to which units to send specific pieces of information, as opposed to percepts always being sent to certain units even when they do not require them (at that time).

In order to improve the performance of the percepts that we do have to send, the Environment Interface Standard (EIS) [17], that we have used as a foundation for implementing our connector, differentiates between three types of percepts[1]:

- **Send once:** this type of percept is only sent once. Such percepts are generally used to send data about the (specific) match when an agent is created, such as information about the map on which the match is played.
- **Send on change:** a percept of this type will only be sent if the percept changes. Such percepts are generally used to update known information, such as a unit's health or the number of available resources.
- **Send always:** a percept of this type will be perceived every time the corresponding agent asks for percepts. Such percepts are generally used to

[1]There are actually four percept types, but we do not consider *on-change-with-negation* as this type will be removed in future versions of EIS due to compatibility issues with knowledge representation languages other than Prolog.

indicate temporary information, such as seeing an enemy unit (which can die, after which the corresponding percept is no longer generated).

Send once percepts will be most performant, whilst send always percepts will be least performant. However, as indicated, some information cannot be represented in a 'more performant' type. It is thus important for to carefully consider which percept category certain (groups of) information would best fit in in order to optimize the performance.

For StarCraft, combining the (finite set of) information that is available through the BWAPI interface with the guidelines as posed in this section lead to a set of about 25 percepts[2]. We have designed and optimized our algorithms to compute the difference between information states in order to generate new percepts as fast as possible. Most percepts are only generated if some change occurred. Our connector has been carefully designed so as to optimize the generation of percepts by first and only once generating the global percepts (i.e., that are not specific to units), such as the list of (visible) friendly and enemy units, followed by the generation of the percepts specific to each entity. This structure also ensures that agents receive their percepts immediately when they ask for them, i.e., they are not generated when requested (which would slow down the agent significantly) but only when information actually changes.

5.4.3. Asynchronous Actions

The actions available for a certain entity define the range of behaviour that is possible for a corresponding agent implementation. The basic design guideline here is that as a rule, *any action that a unit can do* (i.e., that is available in the environment) should be available to its corresponding entity (and thus agent). A unit in StarCraft can roughly choose from about 15 types of actions at any given time. Certain actions are only available to specific types of units (e.g., loading a unit into a loadable building). Some abstractions were used in order to better facilitate the usability of this set of actions for agent programmers. For example, instead of using pixel coordinates, StarCraft allows tile coordinates to be used, i.e., corresponding to a certain block of 32 by 32 pixels (buildings in StarCraft always have a size that is a multiple of 32 pixels in any dimension). This abstraction of pixels to tiles is also used in coordinates in percepts, thus not only ensuring easy compatibility with the actions but also allowing for percepts containing coordinates to be updated significantly fewer times when a unit is moving for example. We also note that BWAPI does not explicitly support grouping units (i.e., as a human player would do), and thus each unit needs to choose its own course of action. However, creating group behaviour in a multi-agent system is facilitated through inherent mechanisms such as messaging between agents. Manager agents thus do not need specific actions from a connector, as they can rely solely on the facilities in the agent platform.

However, as a MAS platform uses and runs agents in its own (set of) thread(s) that need to be connected to the environment, synchronisation issues arise that in

[2]For the full set of percepts and actions that are available, we refer to the StarCraft Connector Manual at https://github.com/eishub/Starcraft/blob/master/doc/Resources/StarCraftEnvironmentManual.pdf.

particular for StarCraft pose a challenge, as StarCraft runs at a specific rate, updating the game logic at fixed millisecond intervals in so called '*match frames*'. In existing (C++/Java) BWAPI bots, the match frame function is used as the starting point (or even single function) for all decision making. In principle, this conflicts with a multi-agent approach in which all cognitive agents run in their own separate (autonomous) thread(s). As a solution, we use several synchronisation mechanisms. First, and most importantly, for each entity all requested actions are recorded (queued). On each match frame call, all queued actions (for all entities) are executed, i.e., 'forwarded' to the corresponding unit in StarCraft itself. Agents have to carefully rely on feedback from the environment (i.e., through percepts) to detect the effect of their actions, or when an action has failed (e.g., because some other action by another agent just used up some resources). A basic understanding of the synchronisation issues is thus needed when developing agents for highly dynamic environments such as StarCraft.

Debugging and Testing

For complex environments such as real-time strategy games and StarCraft in particular, it is also essential to provide a developer with environment-specific visualization tooling that provides easy access to information that will allow the developer to understand what is going on in this environment. Which (types of) tooling can be provided is specific to an environment and the access provided by the basic API made available by the environment. In our case study, we have found that visualization tooling is most useful for providing insight into basic capabilities such as navigation, the status of units, and the progress of long-term actions such as a buildings producing a unit. For example, even though agents do not exercise low-level navigation control, agents do control the setting of target locations where units will move to. We therefore provide a developer with the option to enable visual cues about where a unit is moving to in order to be able to debug the agent code that sets these target locations. Another example of what our connector supports is visualizing when a unit is being produced by a building, removing the need to click on each building to see what it is producing (and how far along this production is) when trying to debug the production logic in a specific building agent. Visualizations like this can be implemented in StarCraft by using its debug drawing features that support drawing lines or writing text in the game window. Using these basic features, our connector allows for specific visualizations to be created by agents themselves (i.e., through calling specific actions), also facilitating drawing custom texts above in-game units. Examples of such 'debug visualizations' in StarCraft are shown in Figure 5.4.

More generally, to be able to debug and test multi-agent systems effectively and efficiently in an environment such as StarCraft where hundreds of agents are running simultaneously, requires a developer to have access to cheats that disclose or even modify gaming information that is not normally available to a player. StarCraft specifically offers useful development functionalities (through BWAPI calls) like removing the fog of war (i.e., making the whole map visible to the player), quickly gaining resources, or to making units invincible. We have integrated these functionalities in a separate development tool (that includes a button for gaining

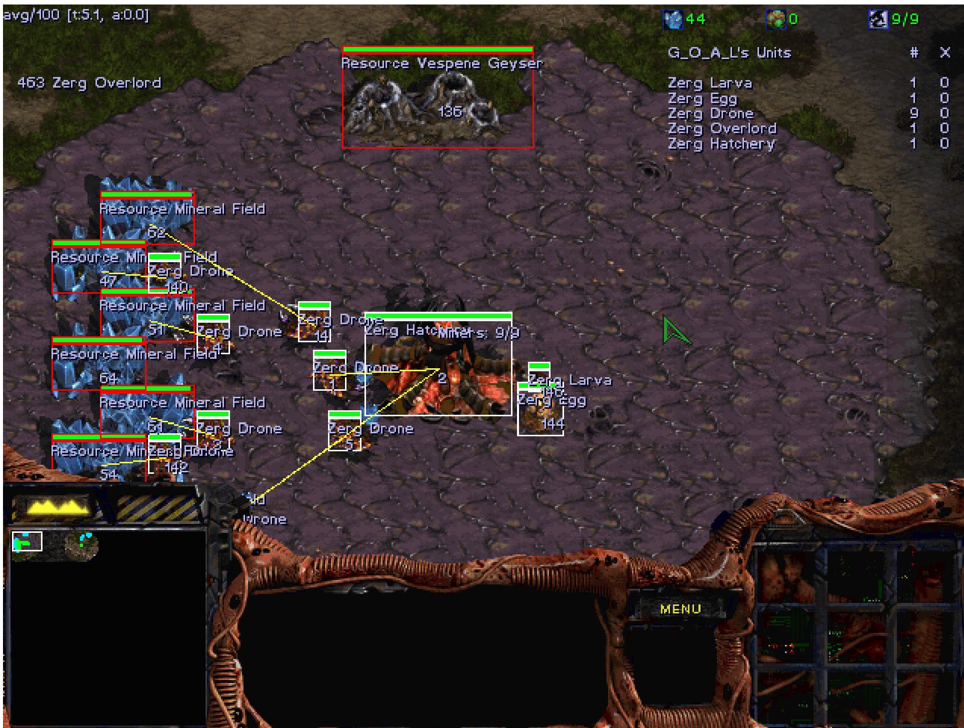


Figure 5.4: A screenshot of StarCraft with a bot performing many debug draw actions.

resources for example) and through initialization properties of the connector (e.g., making units invincible right from the start of a match) in order to make them easily accessible.

5.4.4. Evaluation

As high performance is critical for any cognitive approach that uses many agents to deal with the challenges of AI for RTS, it is important to verify the (CPU) performance of a connector. In addition, one should evaluate the requirement that a connector does not restrict the strategy space in any essential way by for example examining the success (i.e., in tasks in the environment) of a set of cognitive MAS implementations that make use of the connector. We do so by discussing the lessons we learned from the use of our connector by over 500 students in two years.

Performance

Complex real-time tasks, such as effectively attacking enemy units in StarCraft, potentially require a new decision to be made in each match frame (based on the new information such a frame generates). As our approach is based on an unit-agent mapping, there are at least as many agents as units in the game. To be performant,

we need to show that all agents have the opportunity to receive new percepts and make a decision (i.e., perform an action) each match frame. AI tournaments run StarCraft at speeds of at least 50 match frames per second, which implies that in our case every agent should receive new information and be able to perform a new action at least 50 times per second as well, i.e., averaging[3] at most 20 milliseconds for performing all cycles of the agents in a MAS. We assume here that no single agent should perform less than 50 decision cycles per second, even though many agents will not need that many decision cycles (e.g., most buildings would not as the decision making for production is not as time critical as for combat for instance). We aim to demonstrate that the minimum load required in the execution of the StarCraft connector leaves sufficient CPU time for adding the key decision logic in an agent program. We do so for our StarCraft connector by evaluating a simple multi-agent system that keeps producing simple units ('Zerglings') that continuously move to a random location on the map. In addition, all of these units are subscribed to all percepts (i.e., have to process them every decision cycle). A cheat was also enabled to ensure that these units cannot die. In this way, the maximum amount of units that a player can have (which is close to 400) can be reached without being influenced by the enemy in the game. Even though a player is very unlikely to reach this number of units in a game in practice, or to have all units subscribed to all percepts, we aim for our connector to provide sufficient CPU time for strategic reasoning even in this worst-case scenario.

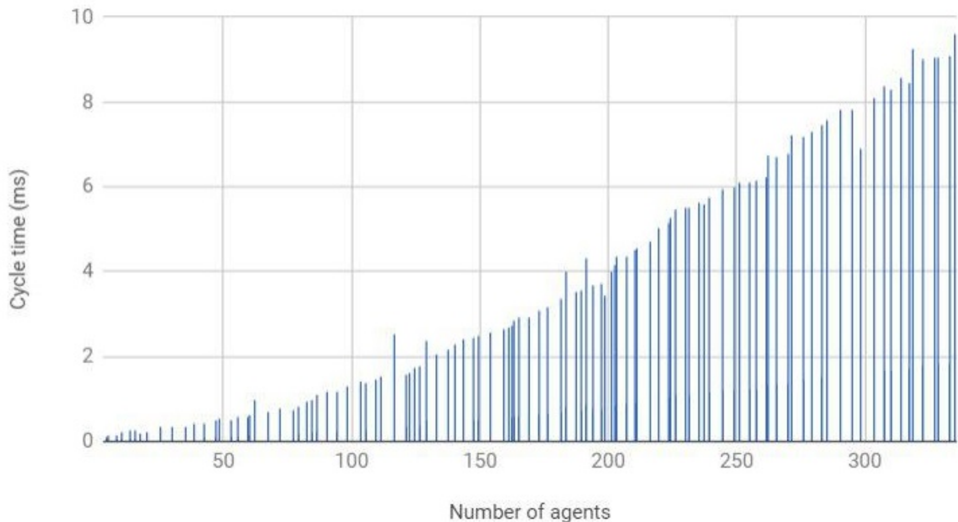


Figure 5.5: The average speed of a decision cycle for all agents under a growing number of agents.

The results of this evaluation for a minimum baseline are shown in Figure 5.5. The evaluation was performed on a system with an Intel i7-6500U CPU and 8GB RAM,

[3]Most tournaments allow bots to take more time for a limited amount of frames during a single match, but we disregard that here.

with the StarCraft game speed set to the default tournament speed of 50 FPS. As each agent runs in its own thread(s), the average time any agent's cycle takes will increase when the number of agents increases due to limited system resources (e.g., the number of available CPU cores). However, even in this worst-case situation with up to 400 agents all processing all information available in the environment, the average cycle time per agent grows to about 10 milliseconds at most. This thus leaves 10 milliseconds for any additional reasoning to be implemented in the MAS in this extreme scenario. In practice, there will be fewer agents that are all subscribed to percepts more selectively. Therefore, in general, we see that around 18 milliseconds (out of the possible 20 milliseconds enforced by the tournaments themselves) will be available to a MAS that uses our connector.

We note that we have designed this baseline MAS such that all of the agents continuously execute decision cycles, whilst in practice, a decision is not required by each agent in every frame. This fact provides further support for our claim that sufficient processing power remains for implementing decision logic, as agents in a MAS with a more diverse set of agents should refrain from executing decision cycles (i.e., 'sleep') from time to time, thus freeing up CPU time for where it is needed most.

Success

As we cannot directly establish whether the full strategy space is made available by a connector, we aim to indirectly determine this by how well a cognitive MAS is able to perform relative to an environment measure that we would like to optimize. For a game like StarCraft, being successful at the game by winning (against other AI implementations) can provide such a measure.

Over the course of two years, groups of students created a varied range of full-fledged StarCraft AI implementations using (different versions of) our connector. After at most 8 weeks of work, nearly all of their implementations are able to defeat the game's built-in AI consistently. Some of the groups joined the Student StarCraft AI Tournament (SSCAIT) [18] with their implementations, successfully competing with the over 100 other active bots (which are mostly written in C++ or Java, frequently based on other well-established implementations, and have often been around for many years or developed by companies like Facebook). One of the students' StarCraft AI implementations that makes use of our connector is currently ranked at around the 50th place with a win-rate of roughly 60%. Altogether, this suggest that we have made the strategy space associated with StarCraft sufficiently available.

During the development and initial uses of the connector, we also gained valuable insights into the benefits and challenges of using current cognitive technologies for engineering complex distributed systems. One particularly challenging development issue that developers face when environments become more complex and the number of agents increases, is that every run of the system will produce different results. For this reason, it is very hard for a developer to test a specific scenario that s/he has in mind without additional tooling to provide a developer with control over the type of scenario that will evolve in the game. This makes testing very

difficult and it thus is of the utmost importance to do whatever possible to provide a developer with tooling and capabilities to handle this. Testing against StarCraft's built-in AI, for example, will give different results on each run. More importantly, it can take quite a while before a scenario of interest occurs (if it does at all). In order to test specific (defined) scenarios, agent programmers should be allowed to *save the state of the game* at any given point, and then load that specific game again at a later stage, which is supported by StarCraft itself. Although our connector has been designed to support such state saving, in practice this will only provide support to some extent, and agent platforms should provide some way of storing and restoring the state of all agents at the same time.

5.4.5. Impact on Cognitive Technology

Even though the StarCraft connector has been optimized as far as possible when it comes to percept delivery, we found that there still are optimizations that can and should only be provided by the cognitive technology that is used, as we can only do so much; if the MAS platform itself is inefficient, it will not be possible to create an effective MAS approach for StarCraft with its strict real-time response requirements. One issue is for example that cognitive agents typically try to run as many decision cycles as possible. Considering the large number of agents that are typically employed in StarCraft, however, this is not ideal. In order to free up cycle time for e.g. agents that that have received new information to reason about. Therefore, we believe that functionalities that reduce the total load on the CPU, such as a '*sleep mode*' in which an agent that does not receive new percepts from the connector or new messages from other agents will not execute any reasoning, should be provided by agent platforms.

However, problems do arise in this mode when for example an agent is supposed to do something (e.g., move around) after it has not received new information for some time. Therefore, a *timing mechanism* should be introduced as well, facilitating the automatic generation of timer percepts upon a certain requested interval (thus waking up the agent after a set amount of time). A *sleep* action can be added as well, allowing a developer to manually sleep an agent for a certain amount of time, and thus free up performance for other agents if they do not need to do any reasoning for a while (even when new information comes in). An example of this is when a building agent starts producing a new unit, and is sure it will keep producing this unit (which takes a while). In addition, to allow developers to get more insight into the performance of their agents, specific logging messages can be added to agents that when enabled, after each decision cycle, show how many queries were performed and how many beliefs, goals, percepts and messages the agent has (received) in total. This can be useful for a developer to for example improve the ordering or nesting of rules in order to reduce the average amount of queries that are executed per cycle, or to keep tabs on the amount of messaging between agents (e.g., one agent might flood another agent with redundant messages due to some bug).

Another observation is that communication with large amounts of agents poses many challenges. In practice, with peer-to-peer based messaging, as is typically

done in cognitive architectures, developers often use broadcasts to all agents in order to prevent having to use numerous bookkeepings of agents, which has an especially large performance impact in systems with many agents (such as those for StarCraft). We believe that this suggests that agent platforms should support a *publish-subscribe* messaging system to be effective, as this prevents agents that need to send messages to other agents from having to deal with continuously keeping track of which agents are relevant for its messages (i.e., interested in the information and still alive). Publish-subscribe messaging facilitates sending messages to a '*channel*'. Agents can subscribe to (and unsubscribe from) such channels, thus receiving messages sent to a certain channel only if they have explicitly indicated they want to do so. This allows for messaging based on content instead of specific targets. This is especially convenient for 'manager agents' to communicate with other (groups of) agents, as such an agent could for instance relay all required information about enemy units in a specific region to a certain channel, to which agents that need that information can then subscribe.

We believe that the application of cognitive agent technologies to complex environments such as StarCraft will yield more ideas for further development.

5.5. Conclusions and Future Work

We have presented a design approach for creating connectors for cognitive agent technology to (complex) environments, illustrated by a case study of such a connector that provides full access to StarCraft. A major challenge that was addressed during the development of this connector was to ensure corresponding cognitive agent systems can be programmed at a high level of abstraction whilst simultaneously allowing sufficient variety in strategies to be implemented by such systems. Based on this challenge, design guidelines for determining the set of available percepts and actions in agent-environment connectors were formulated. The viability of our approach is demonstrated by multiple large-scale practical uses of the StarCraft connector, resulting in a varied set of competitive AIs. Based on the development of the connector and this initial use, we gained valuable insights on the development of complex cognitive agent systems as well, such as the benefits of using publish-subscribe based messaging and the challenges of debugging large sets of agents.

Ensuring a sufficient level of performance of the connector was a significant challenge that had to be addressed in particular in order to demonstrate that a unit-agent mapping (MAS) approach is viable. In our evaluations, we determined the baseline performance of the connector in a worst-case scenario, which shows that on average there remains sufficient CPU time for strategic reasoning in a cognitive MAS. Even though the performance of such a MAS depends largely on the agent technology used itself, we believe that our connector, and thus our design approach, can be effectively used in practice. Although our case study is focused on the 'Brood War' version of StarCraft, the new 'raw API' of StarCraft 2 is reported to be similar to BWPAI by Vinyals *et al.* [19], and tour work should therefore be relatively straightforwardly applicable and/or portable to StarCraft 2 (and possibly other RTS games) in future work.

Finally, through the development and use of our connector for StarCraft, a number of challenges to cognitive agent technologies were identified. One of those challenges is the fact that debugging (cf. Chapter 2) becomes increasingly difficult with increasing numbers of agents. As debugging concurrent programs is a hard problem in general, more work is required in this area; it could for example be useful to visualize the interaction between agents or the CPU time required by each agent. In addition, in order to better support automated testing, (cf. Chapter 3), it may be beneficial to develop a mechanism that automatically saves the state of a MAS when a save game is created in StarCraft. This can be used to immediately initialize a MAS to the desired state when executing a test with a specific save game (i.e., a scenario). Another observation is that communication with large amounts of agents poses many challenges, requiring more investigation in future work, for example into messaging architectures based on a publish-subscribe pattern. Finally, the performance of a MAS itself (i.e., all processing that takes place outside of a connector) is of critical importance in highly dynamic environments such as StarCraft. Functionalities that can reduce the CPU load of a MAS are thus important to explore as well.

References

[1] V. J. Koeman, H. J. Griffioen, D. C. Plenge, and K. V. Hindriks, *Designing a cognitive agent connector for complex environments: A case study with starcraft,* in *Proceedings of the 6th International Workshop on Engineering Multi-Agent Systems*, EMAS '18 (2018).

[2] V. J. Koeman, H. J. Griffioen, D. C. Plenge, and K. V. Hindriks, *Starcraft as a testbed for engineering complex distributed systems using cognitive agent technology,* in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18 (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2018) pp. 1983–1985.

[3] N. R. Jennings, K. Sycara, and M. Wooldridge, *A roadmap of agent research and development,* Autonomous Agents and Multi-Agent Systems **1**, 7 (1998).

[4] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, *Multi-Agent Programming* (Springer, 2009).

[5] J. P. Müller and K. Fischer, *Application impact of multi-agent systems and technologies: A survey,* in *Agent-Oriented Software Engineering: Reflections on Architectures, Methodologies, Languages, and Frameworks*, edited by O. Shehory and A. Sturm (Springer Berlin Heidelberg, 2014) pp. 27–53.

[6] K. V. Hindriks, *The shaping of the agent-oriented mindset,* in *Engineering Multi-Agent Systems: Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, edited by F. Dalpiaz, J. Dix, and M. B. van Riemsdijk (Springer International Publishing, 2014) pp. 1–14.

[7] B. Logan, *A future for agent programming,* in *Engineering Multi-Agent Systems: Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers*, edited by M. Baldoni, L. Baresi, and M. Dastani (Springer International Publishing, Cham, 2015) pp. 3–17.

[8] F. Dignum, J. Westra, W. A. van Doesburg, and M. Harbers, *Games and agents: Designing intelligent gameplay,* International Journal of Computer Games Technology **2009** (2009).

[9] G. Robertson and I. Watson, *A review of real-time strategy game AI,* AI Magazine **35**, 75 (2014).

[10] R. Lara-Cabrera, C. Cotta, and A. Fernández-Leiva, *A review of computational intelligence in RTS games,* in *2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI)* (2013) pp. 114–121.

[11] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, *A survey of real-time strategy game AI research and competition in StarCraft,* IEEE Transactions on Computational Intelligence and AI in Games **5**, 293 (2013).

[12] F. Dignum, *Agents for games and simulations,* Autonomous Agents and Multi-Agent Systems **24**, 217 (2012).

[13] K. V. Hindriks, B. van Riemsdijk, T. Behrens, R. Korstanje, N. Kraayenbrink, W. Pasman, and L. de Rijk, *Unreal GOAL bots,* in *Agents for Games and Simulations II: Trends in Techniques, Concepts and Design*, edited by F. Dignum (Springer Berlin Heidelberg, 2011) pp. 1–18.

[14] B. G. Weber, M. Mateas, and A. Jhala, *Building human-level AI for real-time strategy games,* in *AAAI Fall Symposium: Advances in Cognitive Systems*, Vol. 11 (2011).

[15] A. Heinermann, *Brood War API,* https://github.com/bwapi/bwapi (2008), accessed: 2018-05-12.

[16] A. S. Jensen, C. Kaysø-Rørdam, and J. Villadsen, *Interfacing agents to real-time strategy games,* in *SCAI* (2015) pp. 68–77.

[17] T. M. Behrens, K. V. Hindriks, and J. Dix, *Towards an environment interface standard for agent platforms,* Annals of Mathematics and Artificial Intelligence **61**, 261 (2011).

[18] M. Čertický, P. Paradies, M. Šuppa, B. P. Mattsson, T. Vajda, R. Poniatowski, and S. Klett, *Student StarCraft AI Tournament,* https://sscaitournament.com (2011), accessed: 2018-05-12.

[19] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, *et al.*, *StarCraft II: A new challenge for reinforcement learning,* arXiv preprint arXiv:1708.04782 (2017).

6
Conclusion

The research presented in this thesis focuses on tools for the development of cognitive agents. It provides validated design approaches for source-level debuggers of cognitive agents (Chapter 2), testing frameworks for cognitive agents (Chapter 3), tracers for cognitive agents (Chapter 4), and connectors of cognitive agents to complex environments (Chapter 5). The main question of this thesis is:

How can we support developers of cognitive agents in effectively engineering multi-agent systems?

6.1. Conclusions

From the main question, five research questions were derived, and answered in each respective chapter of this thesis.

RQ 1: *How can we provide developers of cognitive agents with an insight into how observed behaviour relates to the program code?*

We proposed a source-level debugger design for cognitive agents aimed at providing a better insight into the relationship between program code and the resulting behaviour. We identified two different types of breakpoints specifically for agent programming: code-based and cycle-based. The former are based on the structure of an agent program, whereas the latter are based on an agent's decision cycle. We proposed concrete design steps for designing a debugger for cognitive agent programs; by using the syntax and decision cycle of an agent programming language, a set of pre-defined breakpoints and a flow between them can be determined in a structured manner, and represented in a stepping diagram. Based on such a diagram, features such as user-defined breakpoints, visualization of the execution flow, and state inspection can be handled. We provided a concrete design for the GOAL and Jason programming languages, as well as a full implementation for GOAL, and argue that our design approach can be applied to other agent programming

languages as well. A qualitative evaluation shows that agent programmers prefer the source-level (i.e., code-based) over a purely cycle-based debugger.

RQ 2: *How can we automate the detection and localization of failures for developers of cognitive agent programs?*

We proposed an automated testing framework for cognitive agents, facilitating automated failure detection and reducing the debugging effort that is required from a developer. We argued that modules are a natural unit for testing, and associate test conditions with modules of an agent program. We introduced a test language that is used to specify test templates for detecting failure types. These test templates refine a failure taxonomy introduced by Winikoff [1]. A test approach has also been specified that explains how to instantiate test templates and derive test conditions for specific failure types. The main steps of this approach are to (i) define success in terms of functional requirements, (ii) test cognitive state updating, and (iii) classify failures that concern actions and goals.

The proposed test language is minimal in the sense that only two temporal operators are provided. We showed by analysing different agent program samples that the language is nevertheless sufficient for detecting failures in these programs. In about 77% of failures found in this reproduction, the testing framework also pointed to the code location of the corresponding fault. We demonstrated that our approach is not biased towards a specific sample of agent programs by applying the framework to other sample programs, and in a different environment. We were able to adequately detect failures by means of the automated testing framework, i.e., all agents eventually met all functional requirements after fixing the detected failures. We also showed that for single agents, test results are always consistent. However, when running multiple agents, a high number of repetitions might be needed to reproduce the same failure in some cases, suggesting an important direction for future work into the testability of multi-agent systems.

A concrete implementation of the proposed automated testing framework has been performed for the GOAL agent programming language, serving as a prototype for evaluation and as an example for other agent programming languages. Empirical evaluation of a large set of test files and according questionnaires handed in by novice agent programmers confirmed that developers spend a considerable amount of time on testing, reaffirming the importance of proper support for this task.

RQ 3: *How can we facilitate developers of cognitive agents in employing 'back-in-time' debugging techniques?*

We proposed a tracing mechanism design that supports omniscient debugging for cognitive agents, a technique that facilitates debugging by moving backwards in time through a program's execution. Using a prototypical implementation of the tracing mechanism in the GOAL agent programming language, we evaluated and demonstrated empirically that the mechanism is efficient and does not substantially affect the runs of program in the sense that the same failures can be reproduced

when the mechanism is turned on and off. This essentially shows that our mechanism is fast enough and can be used in practice for debugging failures without a need to rerun a program.

We also introduced a trace visualization method tailored to cognitives agents based on a space-time view of the execution history. A developer can navigate this view, evaluate queries on a trace, and apply filters to it to obtain views of only the relevant parts of a trace. Our approach is integrated with a source-level debugger and traces source code locations, which enables a developer to single-step through a program's execution history and facilitates fault localization.

RQ 4: *How can developers of cognitive agents connect their agents to complex real-time environments?*

We presented a design approach for creating connectors for cognitive agent technology to (complex) environments, illustrated by a case study of such a connector that provides full access to StarCraft: Brood War. A major challenge that was addressed during the development of this prototypical connector was to ensure corresponding cognitive agent systems can be programmed at a high level of abstraction whilst simultaneously allowing sufficient variety in strategies to be implemented by such systems. Based on this challenge, design guidelines for determining the set of available percepts and actions in agent-environment connectors were formulated. The viability of our approach is demonstrated by multiple large-scale practical uses of the StarCraft connector, resulting in a varied set of competitive AIs. Based on the development of the connector and this initial use, we gained valuable insights on the development of complex cognitive agent systems as well, such as the benefits of using publish-subscribe based messaging and the challenges of debugging large sets of agents.

Ensuring a sufficient level of performance of the connector was a significant challenge that had to be addressed in particular in order to demonstrate that a unit-agent mapping approach is viable. In our evaluations, we determined the baseline performance of the connector in a worst-case scenario, which shows that on average there remains sufficient CPU time for strategic reasoning in a cognitive MAS. Even though the performance of such a MAS depends largely on the agent technology used itself, we believe that our connector, and thus our design approach, can be effectively used in practice.

6.2. Limitations

One limitation of this work is that all three proposed design approaches for development tools for cognitive agents have been implemented and evaluated in the GOAL agent programming language. Although the approaches are designed for cognitive agent systems in general, and all assumptions are listed for each respective design, this means that each evaluation has been performed on specific sets of programmers (i.e., students at the Delft University of Technology). We aim for this work to inspire the maintainers of other cognitive agent platforms to apply our design approaches to strengthen the development tools in their platforms in the future,

and perhaps also different groups of programmers (e.g., in industry) to adopt the cognitive toolset. A similar limitation applies to our proposed design approach for connectors of cognitive agents to complex environments, as this approach has been applied to only one such scenario (i.e., StarCraft). Although we believe this environment is prototypical, we similarly aim for this work to promote the development of such connectors.

An additional limitation applies to Chapters 4 and 5 specifically. For these works, no (qualitative) user studies have been performed yet. For the omniscient debugger, we believe that a history-based explanation mechanism that can automatically answer questions such as 'why did this action (not) happen' [2, 3] is first required for developers to enjoy the full potential of this tool. Regarding connectors of cognitive agents to complex environments, there is only a handful of people in the world who develop such connectors at this time. We hope that our proposed design approach will empower others to undertake such development as well, thus also increasing the use of cognitive agents as a solution to AI problems.

Finally, even though explicitly aiming at tooling for multi-agent systems from the start, all studies show that working with more than a handful of agents still vastly complicates a large portion of the development process. Although vitally important for cognitive agents, the debugging and testing of concurrent programs in general is an active research topic [4]. Further work is necessary both in general and for MAS specifically to deal with the issues that concurrency introduces to the developers.

6.3. Contributions

In this work, we have contributed to the field of developing cognitive agents in multiple ways. We have empirically investigated the needs of developers of cognitive agents in effectively engineering solutions to AI problems. Based on this, for the AOP community as a whole, we have introduced design methods for the creation of source-level debuggers, automated testing frameworks, omniscient debuggers, and cognitive connectors; all vital tools for engineering MAS. Each tool has been implemented in the GOAL agent platform, making sure the proposed design approaches are feasible in practice and serving both as a prototype for use in evaluations as well as an open-source example for the developers of other AOP solutions[1].

We believe all of this work also enhances the potential of "demonstrating the added value of cognitive agents" [5]. First, empowering developers of cognitive agent systems to effectively debug and test their systems should enhance their potential willingness to employ these technologies; debugging and testing are a large part of the entire development process after all [6, 7]. Second, providing developers with a design approach for developing efficient cognitive connectors to complex environments (like StarCraft) allows AOP to be actually employed for engineering large-scale complex distributed systems. Finally, our empirical results provide concrete examples of the potential of AOP. In total perhaps close to a thousand students made use of the GOAL agent platform over the past four years. The StarCraft connector, for example, was used in conjunction with the GOAL platform

[1]See https://bitbucket.org/goalhub/ and https://github.com/eishub/.

by three distinct sets of students in two years: from a pilot with around 100 students, a first run with over 200 students, to a second run with over 300 students. As discussed in the evaluation of Chapter 5, the results of (most of) these students are a clear indicator of the success of both the GOAL platform (of which all our development tools are a major part) and the accompanying cognitive connector for StarCraft, which in turn demonstrates the advancements made in AOP tooling and cognitive connectors in general by our work.

All students together form another more societal contribution of our work to the education of AI in general. Students were convincingly enthusiastic about the StarCraft projects for example, a project which would not have been feasible without all the contributions of this thesis. A master's student of ours graduated on the subject of GOAL and StarCraft specifically [8]. Moreover, our endeavours sparked attention outside of the scientific community[2].

Perhaps one of the most profound contributions of this thesis can be found in the back-in-time debugger for AOP of Chapter 4. Not only does it show the vast potential of AOP compared to e.g. OO by requiring only a 10% overhead (instead of up to 300%), efficient agent traces can serve as a foundation for vital future challenges. One of these can be found in the field of Explainable AI (see e.g. Miller [9] for a recent overview). Being able to trace the reasoning of an agent in a real-life deployment is vital for future AI, relating to concepts such as responsibility, transparency, and accountability (see e.g. [10]). For cognitive agents specifically, as they derive their choice of action from their beliefs and goals which are stored in the trace, this potentially provides these agents with the capability to self-explain their behaviour in terms of these concepts. However, on top of the behavioural trace data that we can thus now provide in a trace, e.g. Taylor *et al.* [11] identify at least four more knowledge sources required for (end-user) explanations: (i) agent design rationale, (ii) domain knowledge, (iii) display ontology, and (iv) explanation knowledge. Although test conditions (c.f. Chapter 3) might serve as part of a design rationale, employing the knowledge sources in AOP is subject for future work. I have developed a prototype of an 'explaining debugger' (c.f. Hindriks [2], Winikoff [3]) during my stay at the University of Liverpool, for which publications are in writing at the time of finishing this thesis.

6.4. Future Work

The debugging challenges related to rule-based reasoning and agent decision cycles form an important part of this thesis. However, there are more challenges in debugging cognitive agents that still need to be addressed. One of these is the fact that agents are generally connected to environments that cannot be suspended instantly (or at all), which makes it difficult to understand the context of a defect. This is especially the case when dealing with physical environments, e.g., controlling robots like search-and-rescue drones. Simulating environments could be a possible solution for this, i.e., using a deterministic, suspendable and repeatable version of an environment for debugging purposes. However, this is a major challenge,

[2]See https://sscaitournament.com/index.php?action=blog&date=2017-12-25.

6

especially for large and/or uncertain domains. A similar challenge is encountered when using multiple agents, as the inherent randomness of allocating CPU time to agent threads can also cause differences upon each run of the system[3]. These same challenges also lead us to rely on 'runtime verification' of agents, instead of using formal techniques like model checking [12]. Although such techniques provide much stronger assurances on the behaviour of a MAS, they inherently require a specification of all possible inputs (or at least the range they are in) and the exact effect of each action any agent takes. Again, this could only be feasible in a constrained simulation environment. Moreover, the usability of model checking tools for non-trivial domains in e.g. the time it requires to execute the verification will need to be improved as well.

Although as aforementioned debugging concurrent programs is a major problem in any type of programming language, the agent-oriented paradigm entails a number of aspects that might aid in supporting this for multi-agent systems specifically. For example, the fact that the way in which agents communicate is determined by the platform could be exploited for specific visualizations. In addition, grouping concepts such as organizations and roles could help in clustering information for users, especially considering that the amount of information needed for debugging can easily explode in a systems with many agents.

Most programming languages for cognitive agents embed *knowledge representation (KR) languages* like Prolog or a Web Ontology Language (OWL). This introduces an additional opportunity to employ the debugging frameworks that are available for such embedded languages. For example, SWI Prolog debugging tools could be made available through the GOAL debugger in some way. Moreover, some agent programming languages also embed (instead of extend) an object-oriented programming language such as Java. For these languages, designing a debugger that enables developers of cognitive agents to debug their agents specifically (i.e., without stepping into the reasoning engine itself) is a major open challenge.

Our tracing mechanism for cognitive agents focuses on the core aspects of such agents. However, when using additional components like an ethical reasoner or a reinforcement learner, tracing the (implication of the) reasoning in these components is vital as well. Our 'state change-based' design does inherently facilitate adding any important event to a trace, but defining those events in such components is still an open challenge.

Finally, through the development and use of our cognitive connector for StarCraft, further challenges to cognitive agent technologies in general were identified. In time-critical environments, profiling (i.e., measuring the space or time complexity of a program) is a vital development process. However, in the context of multi-agent systems, profiling has not been received much attention so far. It could for example be useful to visualize the CPU time required by each (module of each) agent. In addition, in order to improve the support for automated testing with complex environments without any form of simulation, it would be beneficial to develop a mechanism that automatically saves the state of a MAS when e.g. a save game is created in StarCraft. This could then be used to immediately initialize a MAS to

[3]Only trivial performance-hampering scheduling mechanisms could truly prevent this.

the desired state when executing a test with the specific save (i.e., scenario). Another observation is that communication with large amounts of agents poses many challenges, requiring more investigation in future work, for example into messaging architectures based on a publish-subscribe pattern and organisational structures of multi-agent systems in general[4]. Finally, the performance of a MAS itself (i.e., all processing that takes place outside of a connector) is of critical importance in highly dynamic environments such as StarCraft. Functionalities that can further reduce the CPU load of a MAS are thus important to explore as well.

More generally, the co-existence of a MAS on one side and a connector on the other side raises some questions about the level of abstraction that is desired in each component. It is obvious that an agent should not consist of a single action like `win` which then delegates all tasks to within the connector nor of actions as specific as controlling the individual joints as a robot; however, the grey area that is in-between these two ends perhaps deserves more attention, as developers wanting to tackle a certain problem in a certain with our tools will need to create both the MAS and the corresponding connector for that environment. To this end, a clear overarching design approach would be beneficial.

References

[1] M. Winikoff, *Novice programmers' faults and failures in GOAL programs,* in *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14 (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2014) pp. 301–308.

[2] K. V. Hindriks, *Debugging is explaining,* in *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 7455, edited by I. Rahwan, W. Wobcke, S. Sen, and T. Sugawara (Springer Berlin Heidelberg, 2012) pp. 31–45.

[3] M. Winikoff, *Debugging agent programs with why? questions,* in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17 (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2017) pp. 251–259.

[4] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal, *10 years of research on debugging concurrent and multicore software: A systematic mapping study,* Software Quality Journal **25**, 49 (2017).

[5] K. V. Hindriks, *The shaping of the agent-oriented mindset,* in *Engineering Multi-Agent Systems: Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, edited by F. Dalpiaz, J. Dix, and M. B. van Riemsdijk (Springer International Publishing, 2014) pp. 1–14.

[6] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling, *What do we know about defect detection methods?* Software, IEEE **23**, 82 (2006).

[4] I recently contributed to this line of research with two undergraduate students in Bernstein *et al.* [13].

[7] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009).

[8] T. Peeters, *Evaluating a Cognitive Agent-Orientated Approach for the creation of Artificial Intelligence in StarCraft*, Master's thesis, Delft University of Technology (2018).

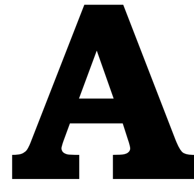
[9] T. Miller, *Explanation in artificial intelligence: Insights from the social sciences,* Artificial Intelligence **267**, 1 (2019).

[10] Ethically Aligned Design, *A Vision for Prioritizing Human Well-being with Autonomous and Intelligent Systems*, Report (The IEEE Global Initiative for Ethical Considerations in Artificial Intelligence and Autonomous Systems, 2018).

[11] G. Taylor, R. M. Jones, M. Goldstein, R. Frederiksen, and R. E. Wray III, *Vista: A generic toolkit for visualizing agent behavior,* in *Proceedings of the 11th Conference on Computer Generated Forces and Behavioral Representation* (Institute for Simulation and Training, University of Central Florida, Orlando, FL, USA, 2002) pp. 157–167.

[12] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini, *Model checking agent programming languages,* Automated Software Engineering **19**, 5 (2012).

[13] B. A. Bernstein, J. C. Geurtz, and V. J. Koeman, *Evaluating the effectiveness of multi-agent organisational paradigms in a real-time strategy environment,* in *Proceedings of the 2019 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '19 (International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, USA, 2019) p. in press.



Source-Level Debugger Questionnaire and Correlation Analysis

The following instructions were given in the questionnaire that was used for the evaluation in Chapter 2:

1. Indicate the total time you spent on programming for this assignment.
   (Less than 8 hours, 8-12 hours, 12-16 hours, 16-20 hours, or More than 20 hours)
2. Indicate the percentage of time that you spent on debugging and testing out of the total time you spent on programming for this assignment.
   (Less than 10%, 10 to 25%, 25 to 40%, 40 to 55%, or More than 55%)
3. Indicate the percentage of time that you spent using the Eclipse debugger out of the total time you (both) spent on programming for this assignment.
   (Less than 10%, 10 to 25%, 25 to 40%, 40 to 55%, or More than 55%)
4. Select the option that best matches you.
   (I use the debugger after an automated test fails, I use the debugger to see how my program behaves, I use the debugger when I see that the robots in BW4T do something wrong, or I hardly use the debugger)
5. Order the following debugging features provided by the Eclipse debugger from most useful to least useful.
   (Logging, Interactive Console, Stepping, Breakpoints, State Inspection, and Watch Expressions)
6. Order each of the following aspects of agent-oriented programming from making debugging more easy to more difficult.

A

(Embedded KR languages like Prolog, External environments like BW4T, Cognitive states, Decision cycles, Rule-based decision making, and Multiple agents)

7. Rate how effective you find source-level debugging for locating faults in an agent program.
(Very Effective, Effective, Somewhat Effective, Not that Effective, Ineffective)

8. Provide any comments you have on agent programming, developing, debugging, and testing. We would like to know what you think works well but would also appreciate any comments or suggestions for improving how you can develop agent programs.

The following provides correlation analyses of the results of the evaluation as discussed in Chapter 2 (*N=94*).
**. Correlation is significant at the 0.01 level (2-tailed).
*. Correlation is significant at the 0.05 level (2-tailed).

Table A.1: Correlation Analysis of the answers to Questions 1, 2, 3, and 7.

Pearson C.	TotalTime	DebuggingTesting	UseDebugger	Effectiveness
TotalTime	1	.436**	.097	.005
DebuggingTesting	.436**	1	.460**	-.071
UseDebugger	.097	.460**	1	.235**
Effectiveness	.005	-.071	.235*	1

Table A.2: Correlation Analysis of the answers to Question 5.

Pearson C.	Logging	Console	Stepping	Breakpoints	Inspection	Expressions
Logging	1	-.043	-.279**	-.129	-.435**	-.216*
Console	-.043	1	-.319**	-.305**	-.248*	-.178
Stepping	-.279**	-.319**	1	-.008	.087	-.335**
Breakpoints	-.129	-.305**	-.008	1	-.269**	-.186
Inspection	-.435**	-.248*	.087	-.269**	1	-.119
Expressions	-.216*	-.178	-.335**	-.186	-.119	1

Table A.3: Correlation Analysis of the answers to Question 6.

Pearson C.	KR	Environments	States	Cycles	Rules	Multiagent
KR	1	.014	-.188	-.438**	-.234*	-.417**
Environments	.014	1	-.249*	-.309**	-.563**	-.054
States	-.188	-.249*	1	-.054	.036	-.330**
Cycles	-.438**	-.309**	-.054	1	.072	-.006
Rules	-.234*	-.563**	.036	.072	1	-.212*
Multiagent	-.417**	-.054	-.330**	-.006	-.212*	1

Curriculum Vitæ

Vincent Jaco KOEMAN

24-01-1992	Born in Hoorn, the Netherlands.

Education

2003–2009 Pre-University Education
Werenfridus Gymnasium, Hoorn
Cambridge Certificate in Advanced English (CAE), CEFR Level C1
International Baccalaureate (IB), English Language A1 Higher Level

2009–2012 Bachelor Computer Science
Delft University of Technology
Economics, Law and Management minor

2012–2014 Master Computer Science
Delft University of Technology
Media and Knowledge Engineering track
Interactive Intelligence specialization

Work experience

2011–2014 Teaching Assistant
Delft University of Technology

2011– Co-founder
Chainels (retailer platform)

2014–2019 PhD candidate Interactive Intelligence
Delft University of Technology
Thesis: Tools for Developing Cognitive Agents
Promotor 1: Prof. dr. K.V. Hindriks
Promotor 2: Prof. dr. C.M. Jonker

List of Publications

14. **B. A. Bernstein, J. C. M. Geurtz, and V. J. Koeman** (2019). *Evaluating the Effectiveness of Multi-Agent Organisational Paradigms in a Real-Time Strategy Environment*. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (in press). International Foundation for Autonomous Agents and Multiagent Systems.

13. **V. J. Koeman, K. V. Hindriks, J. Gratch, and C. M. Jonker** (2019). *Recognising and Explaining Bidding Strategies in Negotiation Support Systems*. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (in press). International Foundation for Autonomous Agents and Multiagent Systems.

12. **V. J. Koeman, H. J. Griffioen, D. C. Plenge, and K. V. Hindriks** (2019). *Designing a Cognitive Agent Connector for Complex Environments: A Case Study with StarCraft*. In Engineering Multi-Agent Systems. EMAS 2018. Lecture Notes in Computer Science (in press). Springer, Cham.

11. **V. J. Koeman, K. V. Hindriks, and C. M. Jonker** (2018). *Automating failure detection in cognitive agent programs*. Agent-Oriented Software Engineering, 6(3-4), pp. 275-308.

10. **V. J. Koeman, H. J. Griffioen, D. C. Plenge, and K. V. Hindriks** (2019). *Designing a Cognitive Agent Connector for Complex Environments: A Case Study with StarCraft*. In Proceedings of the 6th International Workshop on Engineering Multi-Agent Systems. EMAS 2018.

9. **V. J. Koeman, H. J. Griffioen, D. C. Plenge, and K. V. Hindriks** (2018). *StarCraft as a Testbed for Engineering Complex Distributed Systems Using Cognitive Agent Technology*. In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (pp. 1983-1985). International Foundation for Autonomous Agents and Multiagent Systems.

8. **V. J. Koeman, K. V. Hindriks, and C. M. Jonker** (2017). *Omniscient debugging for GOAL agents in Eclipse (Demonstration)*. In Proceedings of the 26th International Joint Conference on Artificial Intelligence (pp. 5232-5234). AAAI Press.

7. **V. J. Koeman, K. V. Hindriks, and C. M. Jonker** (2017). *Omniscient debugging for cognitive agent programs*. In Proceedings of the 26th International Joint Conference on Artificial Intelligence (pp. 265-272). AAAI Press.

6. **V. J. Koeman, K. V. Hindriks, and C. M. Jonker** (2017). *Designing a source-level debugger for cognitive agent programs*. Autonomous Agents and Multi-Agent Systems, 31(5), pp. 941-970.

5. **V. J. Koeman, K. V. Hindriks, and C. M. Jonker** (2016). *Using Automatic Failure Detection for Cognitive Agents in Eclipse (AAMAS 2016 DEMONSTRATION)*. In Baldoni M., Müller J., Nunes I., Zalila-Wenkstern R. (eds) Engineering Multi-Agent Systems. EMAS 2016. Lecture Notes in Computer Science, vol 10093 (pp. 59-80). Springer, Cham.

4. **V. J. Koeman, K. V. Hindriks, and C. M. Jonker** (2016). *Using Automatic Failure Detection for Cognitive Agents in Eclipse (Demonstration)*. In Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (pp. 1507-1509). International Foundation for Autonomous Agents and Multiagent Systems.

3. **V. J. Koeman, K. V. Hindriks, and C. M. Jonker** (2016). *Automating failure detection in cognitive agent programs*. In Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (pp. 1237-1246). International Foundation for Autonomous Agents and Multiagent Systems.

2. **V. J. Koeman and K. V. Hindriks** (2015). *Designing a Source-Level Debugger for Cognitive Agent Programs*. In Chen Q., Torroni P., Villata S., Hsu J., Omicini A. (eds) PRIMA 2015: Principles and Practice of Multi-Agent Systems. PRIMA 2015. Lecture Notes in Computer Science, vol 9387 (pp. 335-350). Springer, Cham.

1. **V. J. Koeman and K. V. Hindriks** (2015). *A fully integrated development environment for agent-oriented programming*. In Demazeau Y., Decker K., Bajo Pérez J., de la Prieta F. (eds) Advances in Practical Applications of Agents, Multi-Agent Systems, and Sustainability: The PAAMS Collection. PAAMS 2015. Lecture Notes in Computer Science, vol 9086 (pp. 288-291). Springer, Cham.

SIKS Dissertation Series

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models

02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language

03 Jan Martijn van der Werf (TU/e), Compositional Design and Verification of Component-Based Information Systems

04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference

05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.

06 Yiwen Wang (TU/e), Semantically-Enhanced Recommendations in Cultural Heritage

07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction

08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues

09 Tim de Jong (OU), Contextualised Mobile Media for Learning

10 Bart Bogaert (UvT), Cloud Content Contention

11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective

12 Carmen Bratosin (TU/e), Grid Architecture for Distributed Process Mining

13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling

14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets

15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval

16 Maarten Schadd (UM), Selective Search in Games of Different Complexity

17 Jiyin He (UvA), Exploring Topic Structure: Coherence, Diversity and Relatedness

18 Mark Ponsen (UM), Strategic Decision-Making in complex games

19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles

20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach

21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems

22 Junte Zhang (UvA), System Evaluation of Archival Description and Access

23 Wouter Weerkamp (UvA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior
25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UvA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TU/e), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent

	49	Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality
2012	01	Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
	02	Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
	03	Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
	04	Jurriaan Souer (UU), Development of Content Management System-based Web Applications
	05	Marijn Plomp (UU), Maturing Interorganisational Information Systems
	06	Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
	07	Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
	08	Gerben de Vries (UvA), Kernel Methods for Vessel Trajectories
	09	Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
	10	David Smits (TU/e), Towards a Generic Distributed Adaptive Hypermedia Environment
	11	J.C.B. Rantham Prabhakara (TU/e), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
	12	Kees van der Sluijs (TU/e), Model Driven Design and Data Integration in Semantic Web Information Systems
	13	Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
	14	Evgeny Knutov (TU/e), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
	15	Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
	16	Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
	17	Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
	18	Eltjo Poort (VU), Improving Solution Architecting Practices
	19	Helen Schonenberg (TU/e), What's Next? Operational Support for Business Process Execution
	20	Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
	21	Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
	22	Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
	23	Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
	24	Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval

25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UvA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques
45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
46 Simon Carter (UvA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
47 Manos Tsagkias (UvA), Mining Social Media: Tracking Content and Predicting Behavior
48 Jorn Bakker (TU/e), Handling Abrupt Changes in Evolving Time-series Data
49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

---

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support

02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UvA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience

29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TU/e), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UvA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UvA), Exploration and Contextualization through Interaction and Concepts

---

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TU/e), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information

15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UvA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TU/e), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data
33 Tesfa Tegegne (RUN), Service Discovery in eHealth
34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
36 Joos Buijs (TU/e), Flexible Evolutionary Algorithms for Mining Structured Process Models
37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital
40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models

	43	Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
	44	Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.
	45	Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
	46	Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
	47	Shangsong Liang (UvA), Fusion and Diversification in Information Retrieval
2015	01	Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response
	02	Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
	03	Twan van Laarhoven (RUN), Machine learning for network data
	04	Howard Spoelstra (OUN), Collaborations in Open Learning Environments
	05	Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
	06	Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
	07	Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
	08	Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
	09	Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
	10	Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
	11	Yongming Luo (TU/e), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
	12	Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
	13	Giuseppe Procaccianti (VU), Energy-Efficient Software
	14	Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
	15	Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
	16	Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
	17	André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
	18	Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
	19	Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
	20	Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
	21	Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
	22	Zhemin Zhu (UT), Co-occurrence Rate Networks
	23	Luit Gazendam (VU), Cataloguer Support in Cultural Heritage

24 Richard Berendsen (UvA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs
33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

---

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UvA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VU), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UvA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UvA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (TU/e), Mining Social Structures from Genealogical Data

20 Daan Odijk (UvA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UvA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UvA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TU/e), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UvA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UvA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UvA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy

	47	Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
	48	Tanja Buttler (TUD), Collecting Lessons Learned
	49	Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
	50	Yan Wang (UvT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains
2017	01	Jan-Jaap Oerlemans (UL), Investigating Cybercrime
	02	Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
	03	Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
	04	Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
	05	Mahdieh Shadi (UvA), Collaboration Behavior
	06	Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
	07	Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
	08	Rob Konijn (VU), Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
	09	Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
	10	Robby van Delden (UT), (Steering) Interactive Play Behavior
	11	Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
	12	Sander Leemans (TU/e), Robust Process Mining with Guarantees
	13	Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
	14	Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
	15	Peter Berck (RUN), Memory-Based Text Correction
	16	Aleksandr Chuklin (UvA), Understanding and Modeling Users of Modern Search Engines
	17	Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
	18	Ridho Reinanda (UvA), Entity Associations for Search
	19	Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
	20	Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
	21	Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
	22	Sara Magliacane (VU), Logics for causal inference under uncertainty
	23	David Graus (UvA), Entities of Interest — Discovery in Digital Traces
	24	Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
	25	Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search

26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TU/e), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
40 Altaf Hussain Abro (VU), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VU), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement
46 Jan Schneider (OU), Sensor-based Learning Support
47 Jie Yang (TUD), Crowd Knowledge Creation Acceleration
48 Angel Suarez (OU), Collaborative inquiry-based learning

2018 01 Han van der Aa (VU), Comparing and Aligning Process Representations
02 Felix Mannhardt (TU/e), Multi-perspective Process Mining
03 Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
04 Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
05 Hugo Huurdeman (UvA), Supporting the Complex Dynamics of the Information Seeking Process

06 Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
07 Jieting Luo (UU), A formal account of opportunism in multi-agent systems
08 Rick Smetsers (RUN), Advances in Model Learning for Software Systems
09 Xu Xie (TUD), Data Assimilation in Discrete Event Simulations
10 Julienka Mollee (VU), Moving forward: supporting physical activity behavior change through intelligent technology
11 Mahdi Sargolzaei (UvA), Enabling Framework for Service-oriented Collaborative Networks
12 Xixi Lu (TU/e), Using behavioral context in process mining
13 Seyed Amin Tabatabaei (VU), Computing a Sustainable Future
14 Bart Joosten (UvT), Detecting Social Signals with Spatiotemporal Gabor Filters
15 Naser Davarzani (UM), Biomarker discovery in heart failure
16 Jaebok Kim (UT), Automatic recognition of engagement and emotion in a group of children
17 Jianpeng Zhang (TU/e), On Graph Sample Clustering
18 Henriette Nakad (UL), De Notaris en Private Rechtspraak
19 Minh Duc Pham (VU), Emergent relational schemas for RDF
20 Manxia Liu (RUN), Time and Bayesian Networks
21 Aad Slootmaker (OUN), EMERGO: a generic platform for authoring and playing scenario-based serious games
22 Eric Fernandes de Mello Araujo (VU), Contagious: Modeling the Spread of Behaviours, Perceptions and Emotions in Social Networks
23 Kim Schouten (EUR), Semantics-driven Aspect-Based Sentiment Analysis
24 Jered Vroon (UT), Responsive Social Positioning Behaviour for Semi-Autonomous Telepresence Robots
25 Riste Gligorov (VU), Serious Games in Audio-Visual Collections
26 Roelof Anne Jelle de Vries (UT), Theory-Based and Tailor-Made: Motivational Messages for Behavior Change Technology
27 Maikel Leemans (TU/e), Hierarchical Process Mining for Scalable Software Analysis
28 Christian Willemse (UT), Social Touch Technologies: How they feel and how they make you feel
29 Yu Gu (UvT), Emotion Recognition from Mandarin Speech
30 Wouter Beek, The ”K” in ”semantic web” stands for ”knowledge”: scaling semantics to the web

---

2019 01 Rob van Eijk (UL), Comparing and Aligning Process Representations
02 Emmanuelle Beauxis Aussalet (CWI, UU), Statistics and Visualizations for Assessing Class Size Uncertainty
03 Eduardo Gonzalez Lopez de Murillas (TU/e), Process Mining on Databases: Extracting Event Data from Real Life Data Sources
04 Ridho Rahmadi (RUN), Finding stable causal structures from clinical data
05 Sebastiaan van Zelst (TU/e), Process Mining with Streaming Data
06 Chris Dijkshoorn (VU), Nichesourcing for Improving Access to Linked Cultural Heritage Datasets

07 Soude Fazeli (TUD),
08 Frits de Nijs (TUD), Resource-constrained Multi-agent Markov Decision Processes
09 Fahimeh Alizadeh Moghaddam (UvA), Self-adaptation for energy efficiency in software systems
10 Qing Chuan Ye (EUR), Multi-objective Optimization Methods for Allocation and Prediction
11 Yue Zhao (TUD), Learning Analytics Technology to Understand Learner Behavioral Engagement in MOOCs
12 Jacqueline Heinerman (VU), Better Together
13 Guanliang Chen (TUD), MOOC Analytics: Learner Modeling and Content Generation
14 Daniel Davis (TUD), Large-Scale Learning Analytics: Modeling Learner Behavior & Improving Learning Outcomes in Massive Open Online Courses
15 Erwin Walraven (TUD), Planning under Uncertainty in Constrained and Partially Observable Environments
16 Guangming Li (TU/e), Process Mining based on Object-Centric Behavioral Constraint (OCBC) Models
17 Ali Hurriyetoglu (RUN), Extracting actionable information from microtexts
18 Gerard Wagenaar (UU), Artefacts in Agile Team Communication

Acknowledgements

This dissertation marks the end of almost 10 years at the TU Delft for me, a decade which I thoroughly enjoyed. Here I would like to thank the people who have supported me during this time, especially during my PhD.

I have known my promotor, Koen, for almost that entire decade. His then brand-new first-year course "Project Multi-Agent Systems" sparked my interest in AI. In any capacity (teaching assistant, master's student, scientific programmer, PhD candidate), I have always immensely enjoyed working with you, both on a personal level as well as in the results we achieved; thank you very much!

I have got to know my other promotor, Catholijn, as the caring head of the group. Thank you for always trusting me and the others in the group, giving words of encouragement or putting up a fight when needed, and for always providing insightful feedback on my work or my future career.

I would also like to thank all my other colleagues of the Interactive Intelligence group, in particular: Chris for helping me start my PhD; Timi for bringing some much needed greenery into our office; Chang for his self-brewed beers; Ilir for the discussions about football and making the summer school in Maastricht way more fun; Mike for a fun BNAIC in Groningen and driving me through The Hague by night; Elie for putting up with my horrible French greetings; Fran for bringing so much more fun into the group and visiting me in Hoorn; Bernd for the dinner that we still have to pin a date for (I take full responsibility); Ding for learning us Chinese culture and beating us at ping-pong; Frank for the discussions about explanations; Thomas for deep discussions about machine learning and football alike; Birna for the pleasant collaboration on the AI courses; Reyhan for her genuine care for everyone in the group; Anita for being the best secretary of the TU; Bart for his care for peregrines and servers alike; Ruud for his tireless techsupport; Wouter for the help on the software and many discussions on the art of programming. And thanks to Aleksander, Elena, Frans, Iulia, Joachim, Joost, Malte, Marieke, Mark, Max, Miguel, Myrthe, Nils, Pietro, Rifca, Rijk, Roel, Rolf, Ursula, Willem-Paul, and perhaps others I forgot to mention; sorry for not knowing you better.

I would also like to thank Jonathan Gratch and Michael Winikoff, with whom I have had the pleasure of collaborating with on some of my work. I am also grateful to have enjoyed a month of working at the Autonomy and Verification Laboratory of the University of Liverpool, for which I would like to thank Louise Dennis and Michael Fisher in particular.

In addition, I would like to thank all of the students who have made use of software produced by my research, and who provided me with invaluable feedback and data. In particular, I would like to thank Danny and Harm, Tom, Buster and Jasper, and Wesley and Cedric for working with me on various projects. I would also like to thank all students who have assisted me as a teaching assistant.

Last, but certainly not least, I would like to thank my family and friends who have supported me throughout this journey. Especially my parents, who have put me through my entire education and always fully supported me with more love and care than I could wish for, and my wife and soon-to-be mother of our child, Rianca, for the unconditional love, making sure I took a much needed break from work every now and then, and bringing so much happiness into my life.

Vincent