Designing a source-level debugger for cognitive agent programs

Koeman, Vincent J.; Hindriks, Koen V.; Jonker, Catholijn M.

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

CrossMark

Designing a source-level debugger for cognitive agent programs

Vincent J. Koeman[1] · **Koen V. Hindriks**[1] ·
Catholijn M. Jonker[1]

Abstract 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 of this paper 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.

Keywords Debugging · Cognitive agent · Agent program · Source-level · Decision cycle · Cognitive state

---

✉ Vincent J. Koeman
   v.j.koeman@tudelft.nl

   Koen V. Hindriks
   k.v.hindriks@tudelft.nl

   Catholijn M. Jonker
   c.m.jonker@tudelft.nl

[1] Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands

🖄 Springer

1 Introduction

Debugging is the process of detecting, locating, and correcting faults in a computer program [22]. 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 [46]. 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 [18].

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 [38]. Single-step execution is based on breakpoints, i.e., points at which execution can be suspended [22]. 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 paper, 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 paper is to propose a design approach that is applicable to programming languages for cognitive agents.

The paper is organised as follows. In Sect. 2, we discuss related work on (agent program) debugging. We also discuss the features of a selection of well-known agent programming languages in some detail, as these are important for the design of a debugger for these languages. In addition, we discuss their current debugging tools. Section 3 presents the design of a source-level debugger for agent programs. We identify design principles and requirements that we use in our design, and features that a source-level debugger for agent programs should have. A concrete design is provided for both the GOAL [17] and Jason [5] agent programming languages. In Sect. 4, the set-up and results of an evaluation that was preformed on the source-level debugger implementation that was created for GOAL are discussed. Section 5 concludes the paper with recommendations for future work.

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.

2.1 Debugging and program comprehension

Katz and Anderson [24] 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, 28,29]. Gilmore [14] 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. [31] conclude that the debugging process is a process of iterative *hypothesis refinement* (cf. [43]). Gathering information to comprehend source code is an important part in the process of hypothesis generation. Lawrance et al. [30] 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. [38]). Similarly, Eisenstadt [12] 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 [30,31]. 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 [43].

Eisenstadt [12] 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. [11]). 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 [31,43] and facilitating navigation of a program's source code at runtime [30]. 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 [22], 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 [38,43].

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 [19,40]. 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. [29]). For example, many languages for programming cognitive agents are *rule-based* [4,39]. In rule-based systems, fault localization is complicated by the fact that errors can appear in seemingly unrelated parts of a rule base [44]. 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's goal —————
1 finished.
```

```
———————————————— 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.
```

```
————— 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 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.
```

Fig. 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*

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 Fig. 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).

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 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.

2.3 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

---

[1] This example uses basic elements from the GOAL language; see Sect. 2.3.3.

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 [9], GOAL [17], and Jason [5] and the Java-based languages Agent Factory [33], JACK [42], Jadex [34], and JIAC [21]. 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.

2.3.1 2APL

2APL aims for the effective integration of declarative and imperative style programming [9]. 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 [10] 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 Fig. 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.

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
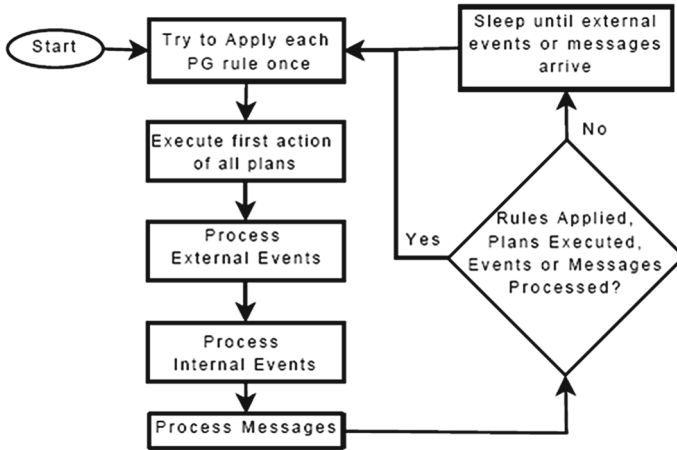
Fig. 2 The 2APL agent decision cycle [9]

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.

2.3.2 Agent Factory

Agent Factory aims to provide a cohesive framework for the development and deployment of multi-agent systems [33]. 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) [3], 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.

2.3.3 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 [17]. 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 Fig. 3 (in Sect. 3.2). 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 [3] 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 paper; 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.

2.3.4 JACK

JACK aims for the elegant and practical development of multi-agent systems [42]. 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
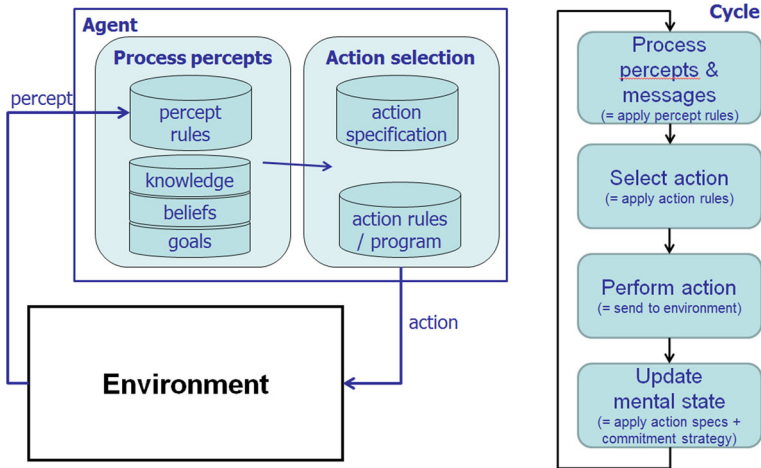
Fig. 3 The GOAL agent structure and decision cycle [17]

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.

2.3.5 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).

2.3.6 Jason

Jason is a multi-agent system development platform based on an extended version of AgentSpeak [36] 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 Fig. 6 (in Sect. 3.3). 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) [37] 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.

2.3.7 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.

2.3.8 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.

3 Design approach for a debugger for cognitive agents

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.

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. [38] 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 [44,45]. This is related to the work Eisenstadt [12], 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 [35]. A debugger should also strive for temporal, spatial, and semantic *immediacy* [41]. 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 Sect. 2, 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 preferred* 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 [7] 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 [32]. 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 [18] and Romero et al. [38] 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 [12] 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. [38] 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.

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 Sect. 2 as well.

3.2.1 Step 1: syntax tree

Inspired by Yoon and Garcia [43], 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. Fig. 4 (top part) illustrates a slightly modified syntax tree for a GOAL agent (see Sect. 2), based on the simplified language specification as shown in Tables 1 and 2 (see Hindriks [20] for the full grammars). Note that '*id*[(*term*)]' represents a call to either a user-specified (environment)

Table 1 The (simplified) core of the GOAL Module Grammar (BNF)

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

Table 2 The (simplified) core of the GOAL (User-Defined) Action Spec. Grammar (BNF)

specification	:=	*useclause*+ *actionspec*+
useclause	:=	**use** *id* [**as knowledge**] .
actionspec	:=	**define** *id*[(*term*)] **with pre**{ *term* } **post**{ *term* }
term	:=	*a (composite) KR expression*

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.

3.2.2 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 Fig. 4, the breakpoints thus identified have been indicated at the corresponding syntax node.
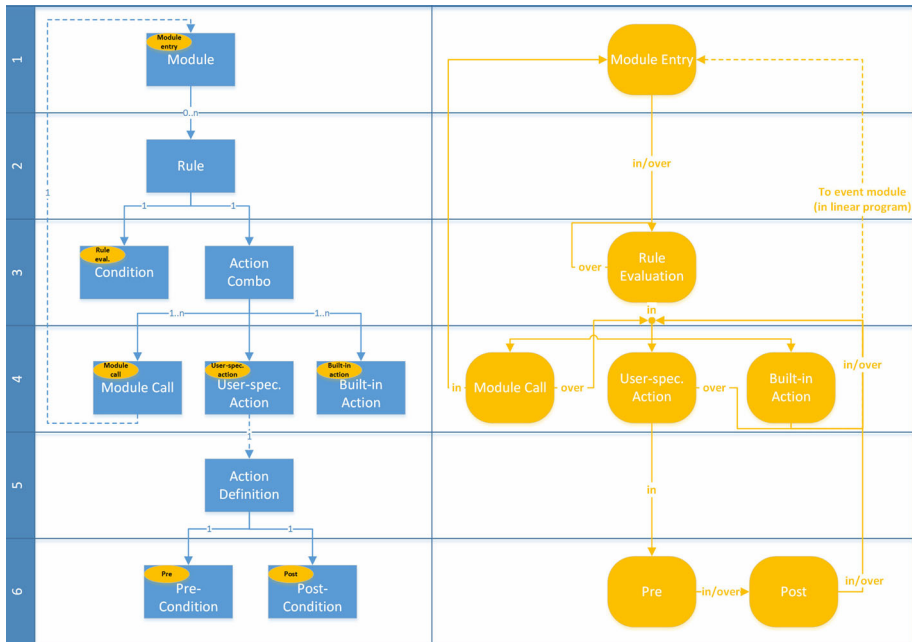
Fig. 4 A GOAL syntax tree with the relevant breakpoints indicated on the nodes that are present at the different levels on the *left side* of the figure, and the stepping flow between those breakpoints (for into and over) illustrated on the *right side* of the figure

3.2.3 Step 2b: cycle-based breakpoints

There are points at which important behaviour occurs that a user wold 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.

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 Fig. 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).

3.2.4 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:
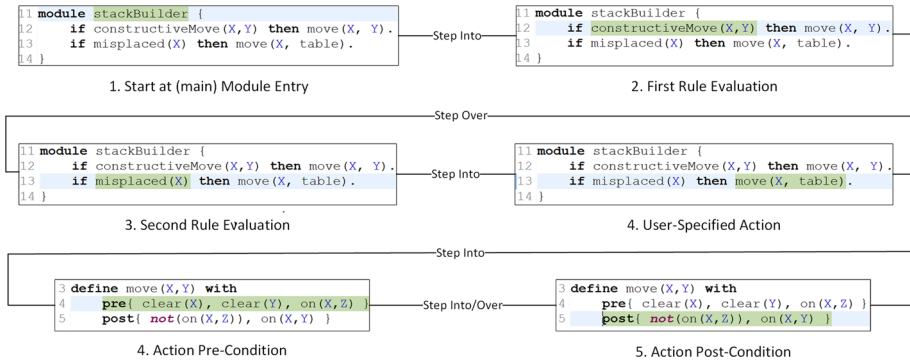
```
11 module stackBuilder {                    11 module stackBuilder {
12    if constructiveMove(X,Y) then move(X, Y).     12    if constructiveMove(X,Y) then move(X, Y).
13    if misplaced(X) then move(X, table).   ─Step Into─  13    if misplaced(X) then move(X, table).
14 }                                         14 }

      1. Start at (main) Module Entry              2. First Rule Evaluation
```

─Step Over─

```
11 module stackBuilder {                    11 module stackBuilder {
12    if constructiveMove(X,Y) then move(X, Y).     12    if constructiveMove(X,Y) then move(X, Y).
13    if misplaced(X) then move(X, table).   ─Step Into─  13    if misplaced(X) then move(X, table).
14 }                                         14 }

      3. Second Rule Evaluation                   4. User-Specified Action
```

─Step Into─

```
3 define move(X,Y) with                      3 define move(X,Y) with
4    pre{ clear(X), clear(Y), on(X,Z) }  ─Step Into/Over─  4    pre{ clear(X), clear(Y), on(X,Z) }
5    post{ not(on(X,Z)), on(X,Y) }          5    post{ not(on(X,Z)), on(X,Y) }

      4. Action Pre-Condition                      5. Action Post-Condition
```

Fig. 5 An example of a stepping flow through a GOAL agent program

- **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 Fig. 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 Fig. 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 Fig. 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 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.

3.2.5 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).

3.2.6 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.

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.

3.2.7 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

Table 3 The (simplified) core of the Jason Agent Specification Grammar (BNF)

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*

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.

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 3 (see Bordini et al. [5] for the full grammar).

The syntax tree (Step 1) based on this grammar is illustrated on the top part of Fig. 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 Fig. 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.

In addition, we look at a stepping flow for the source-level debugging of Jason agents, as illustrated on the bottom part of Fig. 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

---

[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.
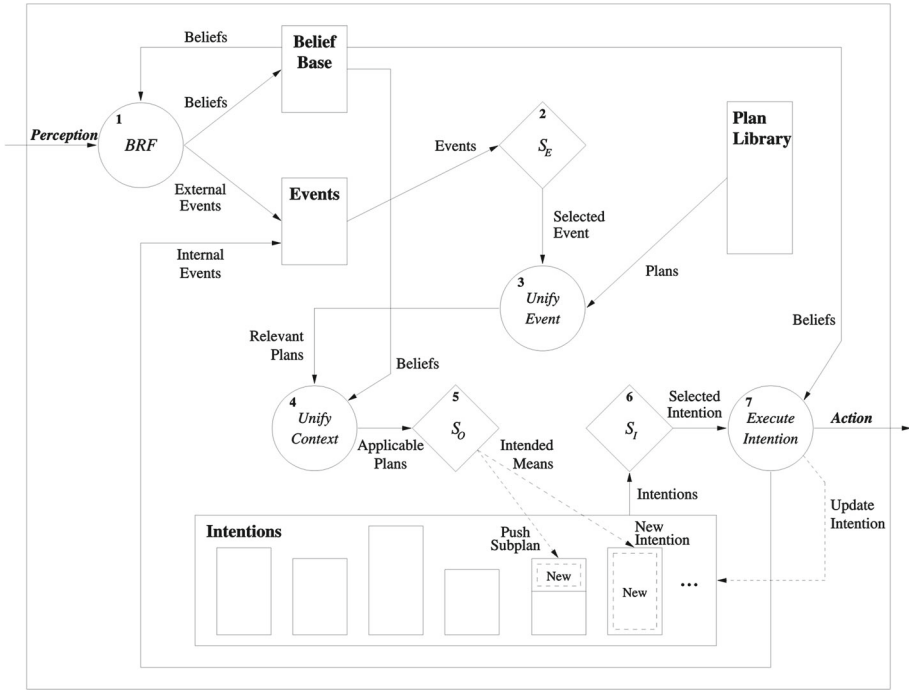
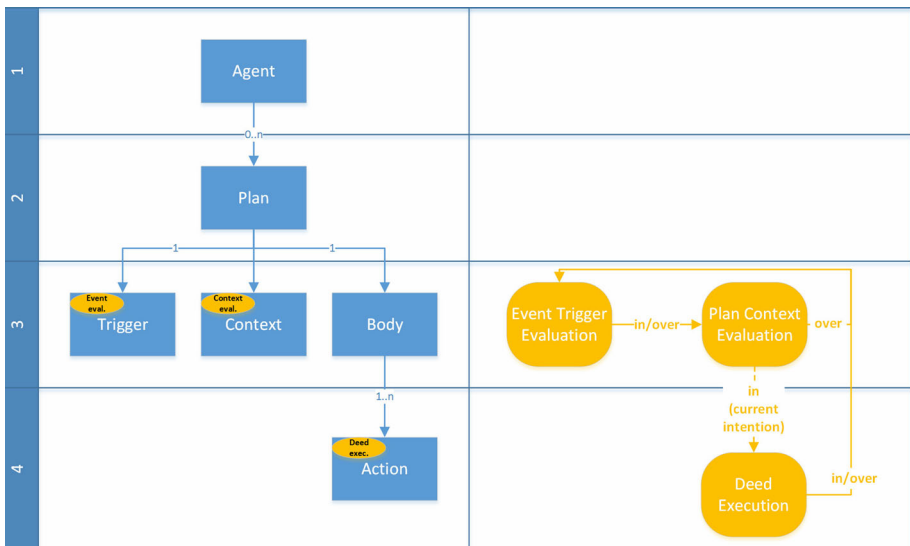**Fig. 6** The Jason agent decision cycle (based on AgentSpeak) [5]

Fig. 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

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 to a user.

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 [26]. The Eclipse platform is based on an open architecture that allows for building on top of well-known existing frameworks [13]. By using Eclipse and the DLTK framework [15], 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, and a testing framework for the validation of agents based on temporal operators has been created [27].

The source-level debugger implementation makes use of the DeBugGer Protocol (DBGP), which is a common debugger protocol for languages and debugger UI communication [6]. 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 Fig. 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 in Sect. 3.2. This implementation was also inspired by the debugging infrastructure of Lindeman et al. [32]. The resulting GOAL debugging interface is illustrated in Fig. 8. In addition, a general overview of the organization of the debugging components is given in Fig. 9. 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

---

[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.
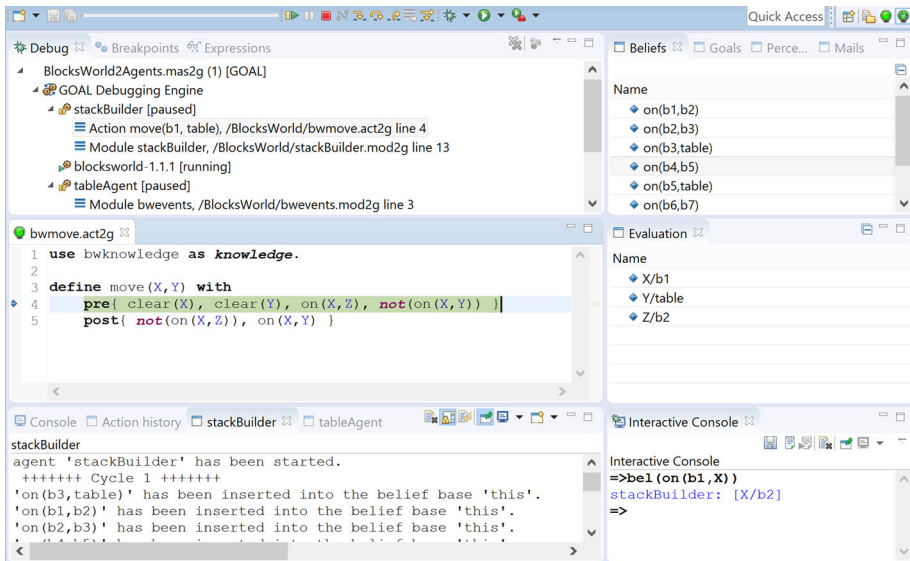
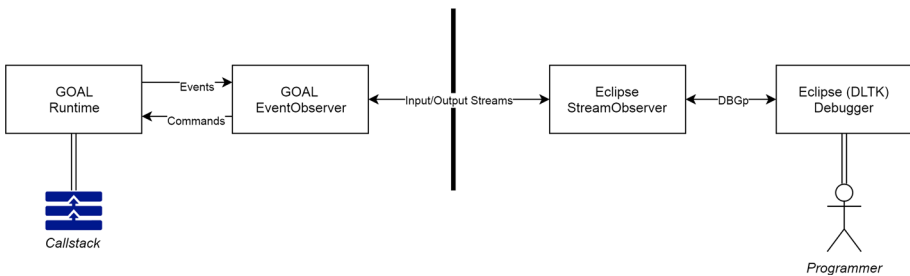**Fig. 8** A screenshot of the GOAL debugging interface in Eclipse Neon

Fig. 9 An overview of the debugging component structure in GOAL (*left*) and Eclipse (*right*)

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.

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 (see Sect. 3.4).

4.1 Quantitative

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 [23]. When handing in their final agents,

Table 4 Descriptive statistics of the performed evaluation ($N = 94$), also see Appendices 1 and 1. 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	" "

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 1.

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 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 Figs. 10, 11, and 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 (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 1. 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
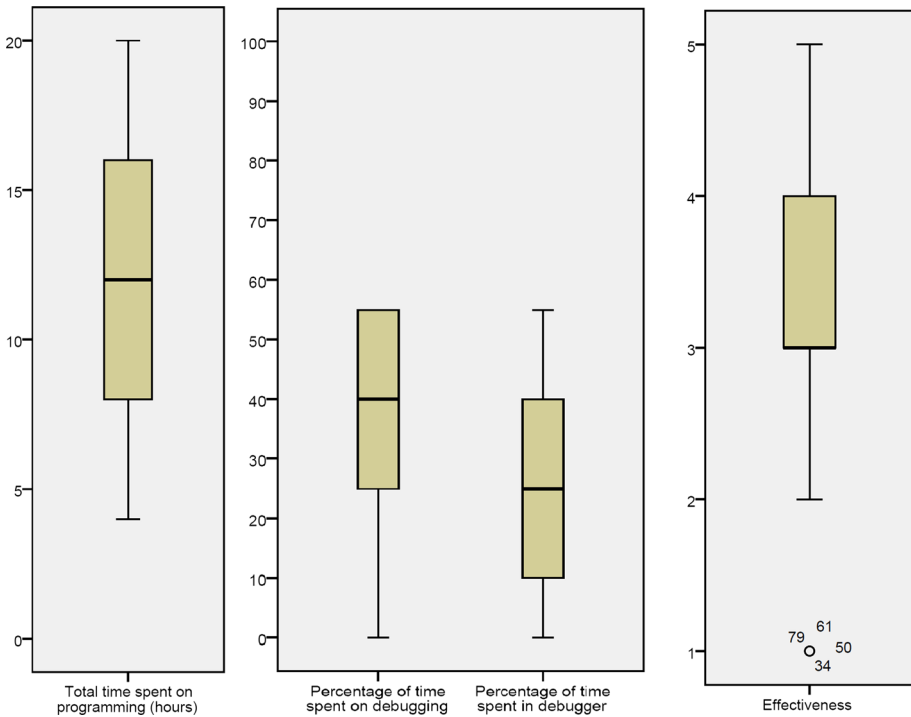
Fig. 10 Boxplots of the results of Questions 1, 2, 3, and 7 (higher means more effective)

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 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.

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:
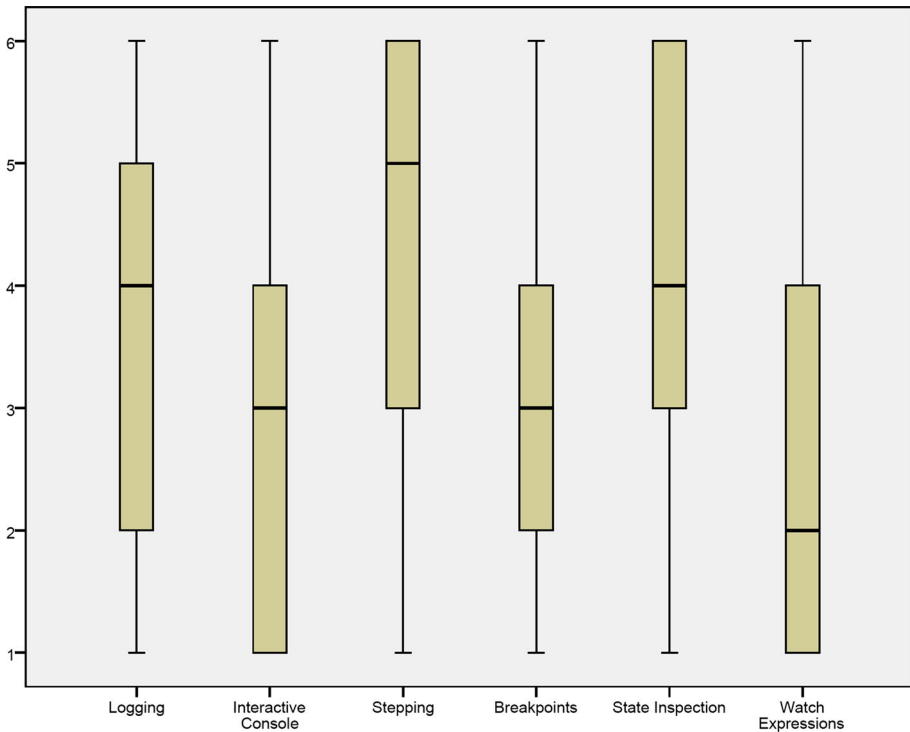
Fig. 11 Boxplot of the results of Question 5 (higher means a more useful debugging feature)

- *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*
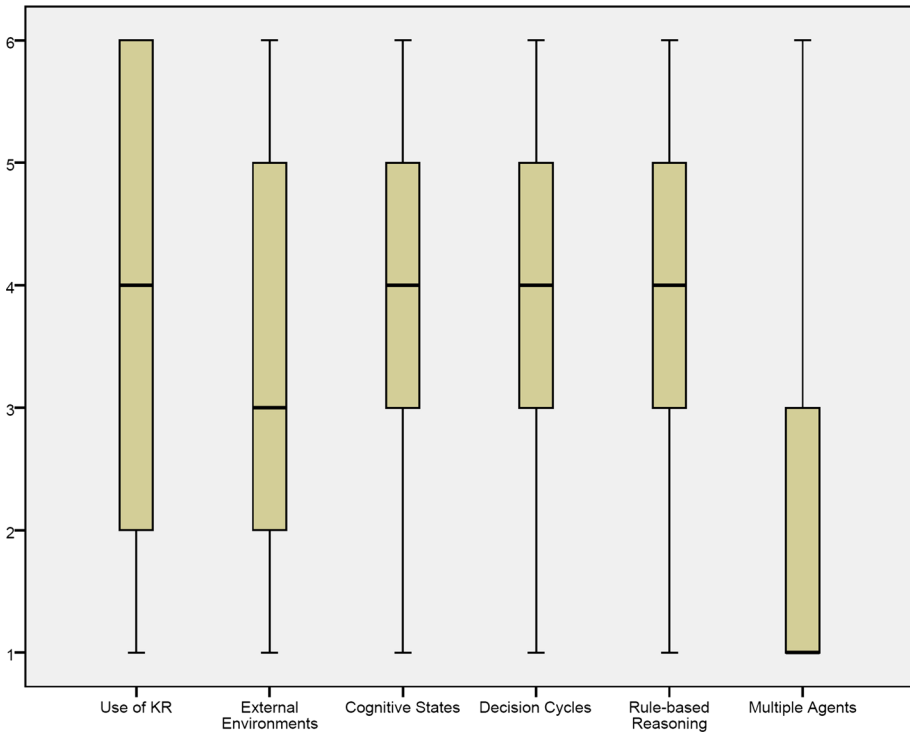
Fig. 12 Boxplot of the results of Question 6 (higher means a more easy AOP aspect)

> *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.

5 Conclusions and future work

In this paper we propose 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 identify 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 propose 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 provide 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 paper. 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 [2], 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 [1,16] 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.

Appendix 1: A Questionnaire

The following instructions were given in the questionnaire that was used for the evaluation in Sect. 4:

1. Indicate the total time you spent on programming for this assignment.
   (Less than 8, 8–12, 12–16, 16–20 h, or More than 20 h)
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.
   (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.

Appendix 2: Correlation analysis

This section provides the correlation analyses of the results of the evaluation as discussed in Sect. 4 ($N=94$) (Tables 5, 6, 7).

Table 5 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

** Correlation is significant at the 0.01 level (2-tailed)
* Correlation is significant at the 0.05 level (2-tailed)

Table 6 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

** Correlation is significant at the 0.01 level (2-tailed)
* Correlation is significant at the 0.05 level (2-tailed)

Table 7 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

** Correlation is significant at the 0.01 level (2-tailed)

* Correlation is significant at the 0.05 level (2-tailed)

References

1. Aldewereld, H., & Dignum, V. (2011). *OperettA: Organization-oriented development environment* (pp. 1–18). Berlin: Springer.
2. Asadollah, S.A., Sundmark, D., Eldh, S., Hansson, H., Afzal, W. (2016). 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Software Quality Journal*, 1–34.
3. Behrens, T. M., Hindriks, K. V., & Dix, J. (2011). Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, *61*(4), 261–295.
4. Bordini, R. H., Dastani, M., Dix, J., & Seghrouchni, A. E. F. (Eds.). (2005). *Multi-agent programming: Languages platforms and applications*. New York: Springer.
5. Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). *Programming multi-agent systems in agentspeak using Jason*. Chichester: Wiley.
6. Caraveo, S., Rethans, D. (2007). DBGP—a common debugger protocol for languages and debugger UI communication. https://xdebug.org/docs-dbgp.php.
7. Collier, R. (2007). Debugging agents in Agent Factory. In R. H. Bordini, M. Dastani, J. Dix, & A. E. F. Seghrouchni (Eds.), *Programming multi-agent systems* (pp. 229–248). Lecture Notes in Computer Science Berlin: Springer.
8. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., & Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, *35*(5), 684–702.
9. Dastani, M. (2008). 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, *16*(3), 214–248.
10. Demoen, B., Tarau, P. (1996). jProlog, a Prolog to Java compiler. https://people.cs.kuleuven.be/~bart. demoen/PrologInJava.
11. Ducassé, M. & Emde, A. M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings of the 10th international conference on software engineering* (pp. 162–171). ICSE '88 Los Alamitos, CA: IEEE Computer Society Press.
12. Eisenstadt, M. (1997). My hairiest bug war stories. *Communications of the ACM*, *40*(4), 30–37.
13. Geer, D. (2005). Eclipse becomes the dominant Java IDE. *Computer*, *38*(7), 16–18.
14. Gilmore, D. J. (1991). Models of debugging. *Acta Psychologica*, *78*(1–3), 151–172.
15. Gomanyuk, S. (2008). An approach to creating development environments for a wide class of programming languages. *Programming and Computer Software*, *34*(4), 225–236.
16. Hannoun, M., Boissier, O., Sichman, J. S., & Sayettat, C. (2000). *MOISE: An organizational model for multi-agent systems* (pp. 156–165). Berlin: Springer.
17. Hindriks, K. V. (2009). Programming rational agents in GOAL. In A. E. F. Seghrouchni, J. Dix, M. Dastani, & R. H. Bordini (Eds.), *Multi-agent programming: Languages, Tools and applications* (pp. 119–157). New York: Springer.
18. Hindriks, K. V. (2012). Debugging is explaining. In I. Rahwan, W. Wobcke, S. Sen, & T. Sugawara (Eds.), *PRIMA 2012: Principles and practice of multi-agent systems* (Vol. 7455, pp. 31–45)., Lecture Notes in Computer Science Berlin: Springer.
19. Hindriks, K. V. (2014). The shaping of the agent-oriented mindset. In F. Dalpiaz, J. Dix, & M. B. van Riemsdijk (Eds.), *Engineering multi-agent systems* (Vol. 8758, pp. 1–14)., Lecture Notes in Computer Science New York: Springer.

20. Hindriks, K.V. (2016). Programming cognitive agents in GOAL. https://bintray.com/artifact/download/goalhub/GOAL/GOALProgrammingGuide.pdf.
21. Hirsch, B., Konnerth, T., & Heßler, A. (2009). Merging agents and services–the JIAC agent platform. In A. E. F. Seghrouchni, J. Dix, M. Dastani, & R. H. Bordini (Eds.), *Multi-agent programming: Languages, tools and applications* (pp. 159–185). New York: Springer.
22. ISO (2010). ISO/IEC/IEEE 24765:2010 systems and software engineering - vocabulary. Technical report, Institute of Electrical and Electronics Engineers, Inc.
23. Johnson, M., Jonker, C., Riemsdijk, B., Feltovich, P.J., Bradshaw, J.M. (2009).Engineering societies in the agents world X: 10th international workshop, ESAW 2009, Utrecht, The Netherlands, November 18–20, 2009. In Proceedings, chap. Joint Activity Testbed: Blocks World for Teams (BW4T) (pp. 254–256). Berlin: Springer.
24. Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, *3*(4), 351–399.
25. Koeman, V. J., & Hindriks, K. V. (2015). Designing a source-level debugger for cognitive agent programs. In Q. Chen, P. Torroni, S. Villata, J. Hsu, & A. Omicini (Eds.), *Principles and practice of multi-agent systems* (pp. 335–350)., Lecture Notes in Computer Science New York: Springer.
26. Koeman, V. J., & Hindriks, K. V. (2015). fully integrated development environment for agent-oriented programming. In Y. Demazeau, K. S. Decker, J. Bajo Pérez, & F. de la Prieta (Eds.), *Advances in practical applications of agents, multi-agent systems, and sustainability: The PAAMS collection, Lecture Notes in Computer Science* (pp. 288–291). New York: Springer.
27. Koeman, V. J., Hindriks, K. V., & Jonker, C. M. (2016). Automating failure detection in cognitive agent programs. In *Proceedings of the 2016 International conference on autonomous agents & multiagent systems* (pp. 1237–1246). AAMAS '16 Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems.
28. Lam, D. N., & Barber, K. S. (2005). Comprehending agent software. *Proceedings of the fourth international joint conference on autonomous agents and multiagent systems* (pp. 586–593). AAMAS '05 New York: ACM.
29. Lam, D. N., & Barber, K. S. (2005). Debugging agent behavior in an implemented agent system. In R. H. Bordini, M. Dastani, J. Dix, & A. E. F. Seghrouchni (Eds.), *Programming multi-agent systems* (pp. 104–125)., Lecture Notes in Computer Science Berlin: Springer.
30. Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., & Fleming, S. (2013). How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, *39*(2), 197–215.
31. Layman, L., Diep, M., Nagappan, M., Singer, J., Deline, R., Venolia, G. (2013).Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In: *2013 ACM / IEEE international symposium on empirical software engineering and measurement* (pp. 383–392)
32. Lindeman, R. T., Kats, L. C., & Visser, E. (2011). Declaratively defining domain-specific language debuggers. *SIGPLAN Notices*, *47*(3), 127–136.
33. Muldoon, C., O'Hare, G. M., Collier, R. W., & O'Grady, M. J. (2009). Towards pervasive intelligence: Reflections on the evolution of the Agent Factory framework. In A. E. F. Seghrouchni, J. Dix, M. Dastani, & R. H. Bordini (Eds.), *Multi-Agent programming: Languages, tools and applications* (pp. 187–212). New York: Springer.
34. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, & A. E. F. Seghrouchni (Eds.), *Multi-agent programming, multiagent systems, artificial societies, and simulated organizations* (Vol. 15, pp. 149–174). New York: Springer.
35. Rajan, T. (1990). Principles for the design of dynamic tracing environments for novice programmers. *Instructional Science*, *19*(4–5), 377–406.
36. Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde & J. W. Perram (Eds.), *Agents breaking away* (Vol. 1038, pp. 42–55)., Lecture Notes in Computer Science Berlin: Springer.
37. Ricci, A., Piunti, M., Viroli, M., & Omicini, A. (2009). Environment programming in CArtAgO. In A. E. F. Seghrouchni, J. Dix, M. Dastani, & R. H. Bordini (Eds.), *Multi-agent programming: Languages, tools and applications* (pp. 259–288). New York: Springer.
38. Romero, P., du Boulay, B., Cox, R., Lutz, R., & Bryant, S. (2007). Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human Computer Studies*, *65*(12), 992–1009.
39. Seghrouchni, A. E. F., Dix, J., Dastani, M., & Bordini, R. H. (Eds.). (2009). *Multi-agent programming: Languages tools and applications*. New York: Springer.

40. Sudeikat, J., Braubach, L., Pokahr, A., Lamersdorf, W., & Renz, W. (2007). Validation of BDI agents. In R. H. Bordini, M. Dastani, J. Dix, & A. E. F. Seghrouchni (Eds.), *Programming multi-agent systems, Lecture Notes in Computer Science* (pp. 185–200). Berlin: Springer.
41. Ungar, D., Lieberman, H., & Fry, C. (1997). Debugging and the experience of immediacy. *Communications of the ACM*, *40*(4), 38–43.
42. Winikoff, M. (2005). Jack intelligent agents: An industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, & A. E. F. Seghrouchni (Eds.), *Multi-agent programming, multiagent systems, artificial societies, and simulated organizations* (Vol. 15, pp. 175–193). New York: Springer.
43. Yoon, B.d. & Garcia, O. (1998). Cognitive activities and support in debugging. In *Proceedings of the fourth annual symposium on human interaction with complex systems* (pp. 160–169).
44. Zacharias, V. (2009). Tackling the debugging challenge of rule based systems. In J. Filipe & J. Cordeiro (Eds.), *Enterprise information systems* (Vol. 19, pp. 144–154)., Lecture Notes in Business Information Processing Berlin: Springer.
45. Zacharias, V., Abecker, A. (2007). Explorative debugging for rapid rule base development. In *Proceedings of the 3rd workshop on scripting for the semantic web at the ESWC*.
46. Zeller, A. (2009). *Why programs fail, second edition: A guide to systematic debugging* (2nd ed.). San Francisco: Morgan Kaufmann Publishers Inc.