A fully integrated development environment for GOAL in Eclipse

The design and development of a mature and professional Integrated Development Environment for the multi-agent programming language GOAL

THESIS

submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

V.J. Koeman

Interactive Intelligence Group
Department of Intelligent Systems
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

A fully integrated development environment for GOAL in Eclipse

Author:	V.J. Koeman
Student id:	4008790
Email:	vj.koeman@quicknet.nl

Abstract

For the GOAL agent programming platform, a new, full-fledged IDE was created that provides support for all phases in the agent program development process, with the concepts of integration and adaptation at its base. Using the DLTK framework, a plug-in for the Eclipse platform was created in multiple iterations, continuously evaluating the usability during this process using SUS evaluations. Besides an editing framework, based on newly developed ANTLRv4 grammars, a fully integrated debugging environment was designed and developed. As agent programming languages are based on very different concepts than the more widely supported and documented programming languages like Java and C, this process was not straightforward. The mapping of popular existing concepts and the creation of new AOP-specific standards has been documented in this work. Besides improving the development support for all different kinds of GOAL developers, this thesis aims to set a new standard in the field of multi-agent programming. Care has been taken to explain all steps in detail, and this work has been made as generally applicable within the multi-agent programming field as possible. Therefore, this thesis is also a full guide to the design and development of a mature and professional IDE for multi-agent programming. The full process of the creation of this IDE is presented in this thesis, with an evaluation of the state-of-the-art through literature and existing IDEs at its foundation. Additionally, a prototype was developed to increase the understanding of the requirements for such an environment.

Thesis Committee

Chair:	Prof. Dr. C.M. Jonker
Supervisor:	Dr. K.V. Hindriks
Committee Member:	Dr. M.B. van Riemsdijk
Committee Member:	Prof. Dr. E. Visser

Contents

List of Figures

List of Tables

Preface

This thesis has been written for my masters program in Computer Science, within the Interactive Intelligence group at the EEMCS department of the TU Delft. After being a teaching assistant in the multi-agent programming courses for several years, I am glad I got the chance to improve the environment that a large group of students uses for developing their assignments. Directly improving their experiences has been a very fulfilling task, and even though there were some bumps in the road, there were always students willing to help me out. Therefore, I would like to extend my gratitude to all of the first year computer science students that have helped me in this process, and sincerely hope the upcoming students will benefit from this work even more. Additionally, I would like to thank all fellow teaching assistants and teachers who assisted me throughout this work.

Of course, all of this would not have been possible without the unconditional support of my supervisor, Koen Hindriks. His extensive and valuable feedback lifted this work to a higher level, and his high level of involvement with my work made working together on this project an unforgettable experience.

Finally, I would like to thank my girlfriend and family for providing the necessary support, even when working all day (or all night) long.

V.J. Koeman
Delft, the Netherlands
August 20, 2014

1. Introduction

The goal of this thesis is to design and develop a mature and professional Integrated Development Environment (IDE) for the multi-agent programming language GOAL. But what is a mature and professional IDE? In order to determine this, some definitions are needed first.

An IDE is a software application that provides comprehensive facilities to computer programmers for software development, consisting of a source code editor, a compiler or interpreter (or both), build automation tools, and a debugger[4]. The most used IDEs are Eclipse, NetBeans, and Visual Studio. Agent-oriented programming (AOP) is a programming paradigm centered on the concept of software agents, as opposed to e.g. objects for the object-oriented programming paradigm[1]. An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors[2]. Multi-agent programming facilitates the use of multiple agents, adding messaging capabilities for example. Multiple AOP frameworks exist, like JADE, Jason, and GOAL. GOAL is a framework for programming rational agents. GOAL agents are based on the belief-desire-intention (BDI) model: they derive their choice of action from their beliefs and goals. The GOAL language is a domain specific language for autonomous decision-making, providing the basic building blocks to design and implement such agents by facilitating the manipulation of an agent's beliefs and goals and structuring its decision-making[3].

Modern software engineering cannot be accomplished without integrated tool support; the use of an IDE should result in greater productivity compared with the use of multiple single-purpose tools for program development, such as a text editor and a separate compiler. When not provided with the support of proper development and maintenance tools, programmers are more likely to waste time and produce low-quality software[6]. The tools that an IDE integrates can be split in horizontal and vertical tools: tools used throughout the development process and tools used in specific phases of the development process, respectively[5]. This structure is illustrated in Fig. 1.1.

A mature and professional IDE provides the required tools in an integrated way, but allows those tools to be easily adapted for use in new contexts as well. Thus, integration and adaptation are key capabilities[7]. Providing those capabilities is no easy undertaking, especially considering the fact that the field of AOP is relatively small in comparison with e.g. object-oriented programming (OOP), operating in a significantly different domain. In this thesis, the design and implementation of an integrated tool set for GOAL will be discussed. Moreover, following the aforemen-

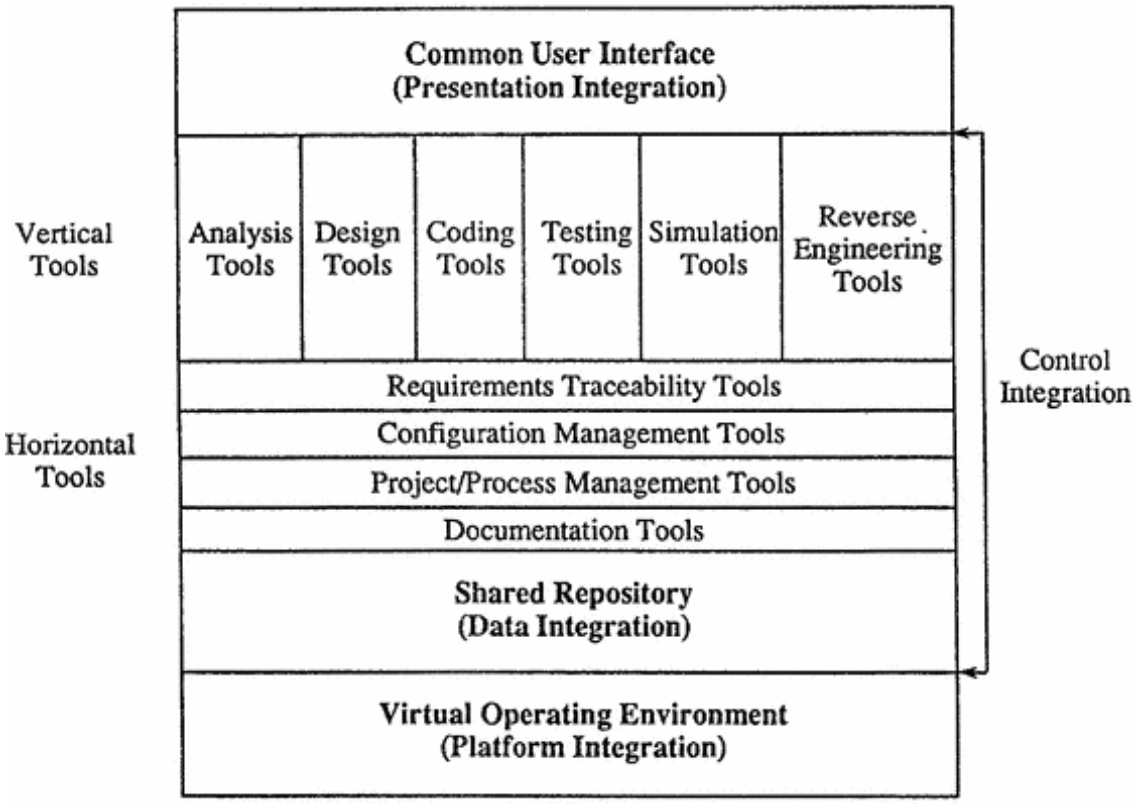

Figure 1.1.: Framework for an Integrated Development Environment

tioned principle of adaptation, this thesis has been constructed as an implementation guide for future AOP IDE developers as well. Through providing a careful analysis of related work in multiple domains, clear requirements, the technical implementation details, and an evaluation afterward, this thesis aims to set the standard for the whole field.

1.1. Problem Statement

GOAL is, among others, used by first-year students at the Technical University of Delft (TU Delft) for a full semester. Although the related courses are generally considered as fun and instructive, the feedback that is given is often of the form: "Nice course, but GOAL feels like a band-aid solution." or "Why is there no stable and complete version of the software yet?". Even though there have been some issues with the GOAL language itself, that have been resolved since, the main 'annoyance' of the students has always been with the accompanying development environment. Probably, this is mainly due to the fact that students compare the GOAL environment with popular IDEs they use in other courses like Eclipse; GOAL is not up to those standards yet, even though it works reasonably well on its own.

A GOAL agent program is a set of modules which consist of various sections including knowledge, beliefs, goals, a program section that contains action rules, and action specifications. Almost all sections are optional, and are represented in a knowledge representation (KR) language such as Prolog, answer set programming, SQL, or the Planning Domain Definition Language (PDDL). The pieces of code in a specific KR language are referred to as 'KR sections'. Currently, GOAL is mainly used with SWI-Prolog as the KR language. SWI-Prolog is a thoroughly developed (since 1987) open source implementation of the Prolog language, and has a very rich set of features[9]. Though GOAL is a Java project, and thus multi-platform by nature, SWI-Prolog only has specific implementations for versions of the Windows, Macintosh, and Linux operating systems. GOAL is actively maintained and developed by the Interactive Intelligence group at the TU Delft since 2006[1].

GOAL agents are executed based on a Multi-Agent System (MAS) file. In such a file, as shown in Fig. 1.2, the optional environment, the relevant agent files, and their launch policies are defined.

```
environment{
   env = "HelloWorldEnvironment.jar".
}
agentfiles{
   "helloWorldAgentEnvironment.goal".
}
launchpolicy{
   when entity@env do launch helloWorldAgentEnvironment.
}
```

Figure 1.2.: An example of a MAS file

In an agent file, different sections can exist within different modules. The init, main, and event modules are built-in modules, but users can also specify their own modules, optionally in an external file. Moreover, knowledge bases (e.g. KR rules) can also be imported from external files. The example agent in Fig. 1.3 prints the text "Hello, world!" ten times using the coupled environment's 'printText' action and 'printText' percept. Actions and percepts are an agent's output and input to and from an environment, and communication amongst agents is also possible. Thus, actions and percepts are effectively an agent's effectors and sensors, defining the agent-environment interaction[10]. Agents are executed by using cycles. Every cycle, the percepts from the environment and messages from other agents are retrieved. Next, the event module's program section is executed, in which the mental state of an agent should be updated using this new information. Next, the main module's program section is executed, or any other user-defined module the agent is currently in (stack). The execution of a module and its program section can be adjusted. The main module's program section, for example, is finished after an action has been executed by default. The event module's program section, in contrast, is always

[1] http://ii.tudelft.nl/trac/goal

fully executed by default. These settings can be changed and used on user-defined modules. The init module is executed before all other modules (when it contains a program section), and is used to define the agent's initial mental state as well. In addition, pre- and post conditions to the actions available in an agent's environment can be provided there: is it possible to execute the action, and how should we update our mental state after the action has been executed.

```
init module{
   beliefs{
      nrOfPrintedLines(0).
   }
   goals{
      nrOfPrintedLines(10).
   }
   actionspec{
      printText(Text){
         pre{ true }
         post{ true }
      }
   }
}

main module[exit=nogoals]{
   program{
      if goal( nrOfPrintedLines(10) ) then printText("Hello, world!").
   }
}

event module{
   program{
      if bel( percept( printedText(NewCount) ), nrOfPrintedLines(Count) )
            then delete( nrOfPrintedLines(Count) ) + insert( nrOfPrintedLines(NewCount) ).
   }
}
```

Figure 1.3.: An example of a GOAL agent file

A first release of the GOAL platform was provided in 2009. The platform includes the GOAL core, an embedded SWI-Prolog build, some example projects and accompanying environments, and a custom-build IDE based on JEdit[2]: a Java-based text editor that was originally released in 1998. Using these tools, one can develop, execute and debug a multi-agent system (MAS), which optionally interacts with an external environment. As mentioned before, the main issues that users face are related to this IDE. Having taken the courses myself, and being a teaching assistant for them in the years after, I have experienced these issues not only from hear-say, but from my own work with the platform. Some often encountered problems are, for example:

- *Inconvenient text editing*: besides not responding to external edits, possibly leading to vanishing code, the text editor itself uses a separate mechanism for

[2]http://www.jedit.org

the syntax coloring that is not always correct. Moreover, the look-and-feel of the editor is not in correspondence with well-known editors (to students), with for example different icons and action-shortcuts being used.

- *Insufficient error detection or recovery*: syntax errors or other possible mistakes in the code are only presented in a single console at the bottom of the screen, for all files together, and only when a file is saved. In special circumstances, Java stack-traces might even be seen there. Moreover, the explanations about the errors are usually not clear to users, often requiring technical assistance for simple syntactic mistakes.
- *Unreliable debugging*: when running an agent, its mental state can be inspected through the so called 'introspector'. However, this is done using basic scroll panes that are completely refreshed every time an update arrives, which results in the view changing multiple times per second. These panes cannot be searched or ordered. Moreover, the relation between the agent's actions, of which logs are available in plain-text consoles, and the written code is not made clear to users. This results in the number one frustration of students being figuring out why their agent is not doing what they think it should do.

These problems give a strong indication of the need for improvement of the GOAL development environment. This environment is already a few years old, and has regularly been updated since. Looking at the repetitive feedback, improving the user experience for a multi-agent programming platform is not a trivial task. Moreover, GOAL and Prolog are examples of rule-based languages: a programming language that works by instantiating rules when activated by conditions in a database. They are also based on logic programming, as the programs are composed of a set of sentences in logical form, expressing facts and rules about some domain. Both of these facts indicate fundamental differences with other, more popular (procedural) programming languages like Java or C, on which most modern IDEs are based, and students have experience with from other courses. There are multiple other factors that add complexity to the tooling that is needed for an agent platform:

- *Embedded KR technology*: making use of a plug-in KR language requires dynamic support for editing, executing and debugging those sections.
- *External environments*: the client/server-like set-up of multi-agent systems and environments adds an additional layer of communication and external dependency.
- *Rule-based evaluation*: providing insight in rule instantiation by conditions in a database is different from providing insight in a procedural execution, requiring different debugging and testing mechanisms.
- *Agent reasoning cycle*: slightly similar to expert systems, the control flow of a GOAL program is based on a reasoning cycle that processes events, evaluates rules, and performs actions, which requires mechanisms different from other languages that use a flow dictated by the code itself.

In this thesis, we will try to identify what is necessary to create a full-fledged agent programming development environment. As an IDE currently exists for GOAL, which is not considered very user friendly, our focus will be the design of an IDE with the aim of improving usability. All steps of the agent program development process will be evaluated using the identified requirements and design guidelines, and supported in a new development environment. Moreover, in an attempt to improve the standards in the whole agent programming field, this will be done in such a way that the whole field can benefit from it by explaining all steps in this process as generally applicable as possible. Considering the formulated problems and their complexity, the research question of this thesis is:

> *What tooling support is needed to support agent program development and what is an adequate design of a development environment for agent-oriented programming?*

Several more specific sub-questions originate from this research question:

- How can we integrate KR technology in an IDE in such a way that one embedded language can easily be exchanged with another?
- What kind of support is needed in an IDE when working with an external environment during execution?
- Which debugging mechanisms are needed when debugging a rule-based language?
- What kind of debugging support should be provided for debugging agent programs that execute reasoning cycles?

In the next section, the approach that was used to answer these questions will be discussed.

1.2. Approach

As a first step, scientific literature on IDEs in general and for agent programming specifically has to be evaluated, together with existing development environments for agent programming. This literature study should be focused on identifying the needs of a developer in all areas of the software development process, in order to obtain requirements for developing a full-fledged development environment. As GOAL is mainly used by students, but by more experienced programmers as well, the differences between these user groups have to be taken into account as well. Furthermore, the state-of-the-art in the field of rule-based languages should be identified, especially by the evaluation of IDEs that follows the literature survey. In this evaluation, IDEs that are either popular in the field or for languages similar to GOAL should be selected and tested in order to obtain more insights into the latest designs. Moreover, the areas in the field that are suitable for improvement are to be identified through this study.

After this research, as an initial case study, a prototype of a modern GOAL IDE should be developed. As the body of literature on agent-programming IDEs specifically is relatively small, this prototype should provide more insight into the functional and design requirements for such an IDE. Multiple potential issues and additional requirements can be identified using this prototype, allowing for a much better design to be made for the actual implementation.

For the final implementation, a set of requirements was identified. These are split in three different steps, reflecting three different phases in the development process:

1. Create a state-of-the-art parsing structure that supports embedded languages properly, e.g. in such a way that any KR language can be supported.
2. Create a framework that will allow user-friendly editing of the different GOAL file types.
3. Create a debugging environment that supports the specific needs of agent programming developers.

For all steps, several required functionalities can be identified. Based on the MoSCoW model[8], the priority of each functionality will be indicated, as listed in Tab. 1.1.

These requirements are part of the evaluation criteria; their fulfillment will be evaluated in the conclusion of this work. In addition to these functional requirements, there are several non-functional requirements as well that will be used to ensure their quality. These requirements are:

1. *Extensibility*: similar programming languages in the field should be able to benefit from the work done in this thesis.
2. *Maintainability*: we should be able to stay up-to-date with the latest developments in the field.
3. *Performance*: the performance should be monitored to avoid a substantial decrease of the platform's power in comparison to the current situation.
4. *Robustness*: the system should be able to deal with errors or mistakes in a user-friendly way, especially considering the large student user base.
5. *Usability*: as increasing the usability of the IDE is the main motivation behind this work, the perceived usability should be monitored constantly.

Using these requirements, the next step is to choose a framework to create our development environment with. Using a proper framework will allow us to benefit from the state-of-the-art without reinventing the wheel ourselves. However, the selection of such a framework should be done carefully, as it should not prevent us from achieving any of the functional requirements, whilst supporting the non-functional requirements in the best possible way. Moreover, the total effort required to create full-fledged IDE will depend heavily on the level of support this framework provides, strengthening the need of careful selection even further.

Parsing structure		
1	M	A parsing structure for all file types (*.goal*, *.mod2g*, *.mas2g*, *.pl*)
2	M	Dynamic support for embedded KR languages
Editing framework		
3	M	Managing projects composed of the supported file types
4	M	A code editor with proper syntax highlighting and error reporting
5	M	The ability to run a system whilst inspecting or logging its output
6	M	Customization to a user's preferences
7	S	An importer for existing (old) projects
8	S	Increased programming support through file templates, auto-completion, automatic indentation, bracket highlighting, and code folding
9	S	Improved support for program comprehension through a source outline and the generation of API documentation
10	S	Improved language learning support through (help) documentation
11	S	The ability to execute a unit-test whilst inspecting or logging its results
12	C	Fully automatic and customizable source code formatting
13	C	Automatic suggestions/fixes for common mistakes
14	C	Advanced refactoring features such as renaming a predicate and all its occurrences in one action
Debugging environment		
15	M	Show the state of all agents in a system and allow running, pausing, or killing an agent.
16	M	Support stepping through the code of an agent
17	M	Allow convenient inspection of an agent's mental state
18	M	Support posing queries to an agent
19	M	Support adding or removing breakpoints during or before execution
20	S	Allow for a continuous inspection of certain aspects of an agent's mental state through watch expressions
21	C	Inspection of an agent's behavior after execution through a navigable history of its decisions and mental states

Table 1.1.: The functional requirements for the GOAL IDE

Based on the selected framework, we will try to fulfill the requirements in the different areas based on their priority. This process has been split into the aforementioned three steps. In the first phase, a parsing structure will be created that supports embedded KR technology in an extendable fashion, e.g. facilitating the support of any KR language as conveniently as possible. In the second phase, the selected framework will be used to create an editing framework. Again, care should be taken to

allow for any embedded KR language to be facilitated easily. Finally, a debugging environment will be created, addressing the different challenges as posed above. As GOAL is a rule-based language based on execution cycles, implementing functionalities such as code stepping is no clear-cut process, requiring careful design.

During and after the implementation, multiple evaluations amongst the users will be done in order to monitor the usability of the platform. After the second and during the third phase, public releases of the IDE will be made, allowing us to continuously collect and integrate user feedback. Finally, using the requirements and the user feedback, the accomplishments of this work will be evaluated, and recommendations for future work will be made.

1.3. Outline

First, a survey of the literature on IDEs and an evaluation of existing multi-agent platforms is presented in sections 2.1 and 2.2. Moreover, because of the relatively small body of work on agent-programming environments specifically, a 'quick-and-dirty' prototype was developed, from which the learned lessons are stated in section 2.3.

Using a bottom-up approach to develop a completely new integrated development environment, frameworks that could be used as a starting point for the development of an IDE for GOAL are evaluated in chapter 3. The next thing to attend to is the GOAL language itself, and thus, in chapter 4, the design and implementation of the language parsing structure is discussed. Next, the implementation of the editing framework is discussed in chapter 5. In chapter 6, the implementation of the debugging environment is discussed. In chapter 7, the development environments for GOAL are evaluated using established approaches for assessing system usability. Finally, in chapter 8, a conclusion is formulated, and recommendations for future work are made.

2. Related Work

In this chapter, first, literature on IDEs in general and for agent programming specifically is evaluated to identify the needs of an agent program developer in all phases of development. Besides this literature, the IDEs of state-of-the-art APL platforms that are either very popular or similar to GOAL are evaluated to obtain more insight into the latest designs. Finally, the prototype that was developed will be discussed, identifying additional requirements or design criteria for the eventual implementation.

2.1. Literature

Early research on development environments by Apple[11] already tries to study the usability of a development environment as a whole. Several development cycles are identified and evaluated for different programming languages. The basics of the 'flexible pane windows' that are used very commonly now-a-days is evaluated here. A main conclusion of this work is that it is difficult to change the way people do their work, and thus development environments should adapt to their users as much as possible. However, many different kinds of users exist, which poses significant issues. Later research on this topic[4] identifies some very noticeable distinctions between novice and experienced users in several development environments for different programming languages. It is noted that there is often a lack in the application of Human Computer Interaction (HCI) principles in the design of IDEs. Applying these principles is hard, because IDEs are abstract and complex systems that have a large amount of very different functionalities bound together in one place. In order to apply HCI principles to an IDE properly, the differences between novice and experienced users that were also found have to be taken into account. For beginners, typical errors are usually not made clear, hierarchical structures are hard to identify, and cognitive overload is often a problem. On the other hand, a lack of relevant and understandable on-screen information is a main problem for more experienced users. The research concludes with the notion that "developers should be provided with IDEs that offer functionalities in more rational, less visually complex formats that reinforce the relation between a specific functionality and the software artifacts on which that functionality acts". IDEs should be made human-centric instead of functionality-centric. Research on this topic does exist, for instance in the field of IDEs for the pedagogic (educational) use of them, but these concepts have not been

widely adapted into industry-standard IDEs[13, 14, 15]. The requirements elicited in these studies can be used to improve this adaptation in our plug-in.

An example of a rule-based language that tries to integrate the differences between users of different experience is DrScheme, a programming environment for the Scheme language[19]. This environment focuses on students and provides four different 'language levels' that all add some more advanced features to the language itself. Other notable features are interactive expression evaluation, clear error reporting, visually enhanced code stepping, and static debugging (evaluation without full execution).

Another topic of interest is the actual use of IDEs by developers. Research has been done on the Eclipse platform in order to determine what views, commands and plug-ins are used the most[16]. Some results of this research include the fact that renaming is done a lot in Java refactoring, followed by moving and extracting functions, and that the current variable instantiations are very important whilst debugging. Finally, key-binds are frequently used by most developers, whilst context menu's in the editor itself are not.

There are some IDEs for rule based languages specifically. For instance, SystemT is a rule-based information extraction system for which a custom IDE has been created[17]. For this IDE, care has been taken to support all three repeating phases of development (develop, test, and analyze) properly. Besides editing, several custom tools have been created in order to support the developer possible, especially by different kinds of visualizations. Similar work exists in the field of answer set programming. In ASPIDE[18], besides the default editing features such as code coloring, auto-completion, refactoring, outline views, (on-line) error highlighting, fix suggestions and code templates, a dependency visualizer, visual code editor, and custom execution result visualizations have been created. In addition, debugging, profiling, and (automated) testing is fully supported, all embedded in one graphical interface. Noticeable is the extensive use of existing libraries for all kinds of purposes (JGraph, Spock, etcetera). In addition, the research that lead to the creation of this IDE involved an extensive study of other rule based development environments, which resulted in the identification of many features logic-based IDEs have. Additional research on development environments for logic programming specifically confirm these requirements, and group them into three categories: development, analysis, and debugging[6]. Combining these works allow us to list the requirements for our IDE in each of those categories.

Development

- A code editor, including:
  - syntax coloring
  - parenthesis or other token pair highlighting
  - undo/redo

  - find/replace
  - quick fixes
  - auto completion
  - code templates (with dynamic definition of them)
  - code annotations
  - automatic code indentation
  - text hover for quick info of variables/predicates
- Management of files on a per-project basis, including:
  - multiple projects
  - multiple files and (sub)folders per project
- Refactoring of variables/predicates.

The research also indicates that it was very evident that Eclipse-based systems had much more ease in the development of text editing and project management features, resulting in a higher usability.

Analysis

- Syntactic or semantic error/warning highlighting (in-code).
- A global error console.
- A content outline for the current file.
- Code documentation.
- A command line interface (for running a system).
- Textual, user friendly result visualization (e.g. system output).
- A profiler.

Program analysis is not restricted to the (static) analysis of source code only; specifications and executions are also interesting sources of data. Moreover, a profiler can give insight into where a program consumes most of its time or space.

Debugging

- A debugger.
- A test suite.
- Configuration/customization of an execution.

Tree main strategies for debugging are identified: verification with respect to specification, checking with respect to language knowledge, and filtering with respect to the error symptom. The verification strategy compares the actual program with some specification of the intended program, the checking strategy looks for suspicious places which do not comply with some explicit knowledge of the programming language, and the filtering strategy filters out parts of the code which cannot be responsible for the error symptom. A manual or automatic (e.g. unit tests) debugging tool needs to support these three strategies.

More general research on the usage of some of the mentioned features in practice exists as well[20]. First, the used development methodologies in the field of logic programming were evaluated. It turned out that most did not use any methodology at all, followed by agile or waterfall/specification methods. Half of the respondents use a full-fledged IDE, but simple text or (graphical) rule editors are also used a lot. Verification, validation, and tool support turned out be the most important hindrances. In addition, whilst comparing rule-based development with 'conventional' development, ease of debugging and tool support were also mentioned as negative factors. On the positive side, rule based systems were judged to be better in almost all other questioned areas. It is also identified that the main difficulty in debugging might be due to either a lack of refined tools or the intrinsic language properties. Moreover, moving to more standardization could lead to better tool support, as the resources and the potential market would grow. Finally, in this light, IDEs should also take care in supporting agile methodologies properly.

As mentioned before, debugging is especially important but also especially hard in rule-based settings. Specific research in this area exists[21]. First, some important intrinsic language issues are identified, mainly related to terminology, opacity, interconnection, error-reporting, procedural debugging, and tool support. Next, four related core principles are derived: interactivity, visibility, declarativity, and modularization. An ideal scheme of integrated test, debug and rule creation is presented, which is different from traditional development because test data is used throughout editing for immediate feedback, and debugging is always available to support rule creation and editing, even in the absence of test queries. In addition, anomaly detection heuristics, rule base visualizations, and explorative debugging support are suggested features. An explorative debugger is a rule browser that shows the inferences as a rule enables, visualizes the logical/semantic connections between rules and how rules work together, supports navigation along connections, allows to further explore the inference a rule enables by digging down into the rule parts, and is integrated into the development environment so it can be started quickly to try out a rule as it is formulated. All these notions should allow rule based systems to reach their full potential.

2.2. State of the art APL IDEs

In the following list, existing IDEs for (multi) agent programming will be evaluated in order to identify the current state of this field. Next, issues of these environments will be discussed in order to learn what to do (and what not) in the IDE that is to be created for the GOAL agent programming language.

2APL[22] uses an Eclipse plug-in created with Xtext: an Eclipse framework that aids in the development of support for domain specific languages. However, this is for editing only, as a separate execution platform exists. Supported editing features include syntax coloring, project management, and auto-completion. Prolog is used as an embedded knowledge representation language, and coloring for this does not seem to be supported. Launching an agent directly from Eclipse is not possible. Simple action logs and mental state tracing functionality (full textual history of each step) are provided in the separate runtime platform. Plain text fields are used to display state information there, for which sorting or filtering is not supported.

AgentFactory[23] is created as a platform with an accompanying Eclipse plug-in that is capable of syntax coloring and simple project management. Again, a separate execution platform is provided, which is very similar to the platform 2APL provides. Around ten separate tabs exist from which plain text status information can be obtained for a single agent at a time. Back-stepping through an agent's mental state history is possible. The debugging tool has been created in an extensible way, allowing it to be customized to support different agent architectures, but the whole debugger is stated to be in an 'embryonic state'.

Jason[24] has plug-ins for Eclipse and JEdit for editing, supporting syntax highlighting and project management. A separate runtime environment ('mind inspector') is available. Most output of this environment uses a plain console, which can be logged as well, but somewhat more advanced visualizations for each round the agent has performed (cycle history) also exist. For example, the communication between agents can be examined through a presentation that is similar to a (UML) sequence diagram.

Jack[25] is a Java-based framework that provides a custom-made IDE. A separate agent run-time is provided, but in addition, graphical tools for debugging such as the tracing of execution plans and inter-agent message passing are also present in the custom IDE. Creating designs and plans using an advanced graphical interface in an UML-like fashion is possible as well. In addition, an Eclipse plug-in with basic editing capabilities for the Plan Language has been created. Unlike most other platforms which are open source, this is a commercial product.

Jadex[26] is an XML-based agent programming language. Existing XML-editing platforms (such as Eclipse) are re-used, with a separate execution and debugging platform. Breakpoints and a textual state history are provided in a basic manner, in addition to a basic agent state inspector that lists the goals, plans and communication and such, without any extended capabilities like sorting or filtering. A flowchart of the communication between the agents can be generated. Debugging can be done whilst stepping through an agent's cycles by using a view that shows a hierarchical display of the currently processed actions.

JIAC[27] is a multi-agent architecture that aims at improving the ease of development and the operation of large-scale, distributed applications. All of this is based on the Eclipse platform. Java and XML are used for this architecture, thus editing comes 'for free'. Next to this platform, the 'Graphical Monitoring Tool for Distributed Agent Infrastructures' (ASGARD) is provided. The ASGARD website states that common methods such as log files, debug outputs and step-by-step execution are not appropriate for most (distributed) multi-agent systems. To this end, a graphical method for monitoring and demonstrating these systems is provided using live 3D visualizations in which the agents from the different systems and their communications and migrations are shown. Specialized views for e.g. system load can be created using plug-ins as well. Moreover, agent states can be visualized and manipulated in the same interface. Their research gives indications that a developer's overview at runtime is vastly improved by this tool.

Jess[28] is an expert system programming that is advertised as having an advanced graphical rule development environment based on Eclipse. Although not an agent programming language, Jess is rule-based and has a mature IDE that is worth evaluating. Its editing features include syntax highlighting, content assist, quick fix assistance, in-code error checking and highlighting, automatic code formatting, an outline view, parenthesis matching, online help through code-hovering, and coupling with the Jess run-time platform for execution. In addition, a custom graphical debugger is provided. In this debugger, stepping through the code is possible with displays of the current call stack and stack frame variables. Setting breakpoints on specific lines of code are supported as well.

Evaluation In Tab. 2.1, an overview of the features the above platforms provide can be seen. For each platform, its development method (Open Source or commercial) is noted. Next, the framework the IDE is based on, if any, is stated. In the editor column, a difference is made between providing just syntax highlighting or more advanced features like auto-completion, a source outline, etcetera. The next two columns highlight the features of mental state inspection. The final column displays the debugging features the platform offers to a developer. All platforms' agents

are based on reasoning cycles, and thus the stepping is as well; none of the AOP platforms provide a code-based stepping mechanism.

Name	**IDE**	**Editor**	**State inspection**	**State history**	**Debugging**
2APL	Xtext (Eclipse)	Advanced, but without KR-highlighting	Textual	Per step	Pausing and stepping agents, logging deliberation steps
Agent Factory	Eclipse	Basic highlighting	Textual	Per step	Pausing and stepping agents
Jason	Eclipse or JEdit	Basic highlighting	Textual	Per step, including communication visualization	Pausing and stepping agents, logging states
Jack	Custom-made or Eclipse	Advanced, including a graphical editor	Textual	Per step, including communication and plan visualization	Pausing and stepping agents, dumping states
Jadex	None (XML editor of choice)	-	Textual	Per step, including communication and action visualization	Pausing and stepping agents
JIAC	Eclipse	Default Java/XML capabilities	Visual	None	3D visualization of agents and their communication, including a mental state editor
Jess	Eclipse	Advanced	N/A	N/A	Code stepping integrated in the IDE

Table 2.1.: An overview of the features of the state of the art APL IDEs

The current state of the field in relation to the guidelines discussed earlier does not seem to be that well. For instance, as aforementioned, none of the AOP platforms provide any way of debugging programs in relation to the actual code. Moreover, with the exception of JIAC, mental states are only presented through plain text

fields that have no advanced features, making them especially hard to use when large numbers of objects are present in a state. In many cases, the code editor is not up-to-date either, with features such as auto-completion missing. In addition, the execution platform or state inspection is usually separated from the editing environment. This does not only mean that a different interface is created every time, but a large amount of windows will have to be used simultaneously by a debugging developer. Significant room for improvement thus exists. Additional research on this subject[29], which was also used for checking the evaluations in the previous paragraph, identifies issues in this field as well. However, as no common ground with respect to the agent programming languages and/or architectures exists, the IDEs are all specific to a certain framework. This scattering does not improve the usability of the agent programming platform as a whole; a user has to use a different environment for each language.

A similar development has taken place in the field of Prolog programming, in which many different styles and thus IDEs exist. Some of these IDEs are noticeably more evolved, with for instance ProClipse[30] and ASPIDE standing out. Although different 'Prolog flavors' exist, they still have their common grounds, which can be said for agent programming as well. Thus, the diversity of the languages should not be impairing. Instead, it motivates this and future work even more, as improvements in one language in the field will almost certainly motivate and inspire others to improve as well. In addition, taking the existence of these common ground into account in advance, steps can be taken in order to allow the whole field to benefit from improvements in a specific language. No one in the field has fully covered all aspects of a modern IDE yet, and the whole field might improve by the creation of a mature agent programming IDE.

2.3. Prototype

As there is insufficient previous work on the development of an agent programming IDE available, a prototype of a GOAL plug-in was created for NetBeans in order to increase the understanding of the requirements for such a plug-in. NetBeans was chosen as the target platform here because of personal experience with it. In this section, its implementation will be explained shortly, after which the learned lessons will be discussed. It has to be noted that nearly no changes to the GOAL core itself were made for this prototype, in the sense that the existing classes before making the plug-in were not modified.

Implementation To register the plug-in, NetBeans uses a module manifest. This file determines the plug-in's identification (including a version number) and links to the 'layer file': an XML file that registers the different functionalities of the plug-in. For instance, for each supported file type, a node in the layer file exists,

providing an image and linking to a specific Java class. This Java class, in turn, can contain a variety of annotations that specify additional options, for instance which file to use as a template when creating a new file of that type. NetBeans facilitates many functionalities automatically by providing classes to extend to. For instance, the class that provides an editor for MAS files just contains two functions: one that returns the class responsible for transforming code parts of a specific type to a category NetBeans can understand (for coloring) and one that sets a human-readable name for the file type.

To provide syntax highlighting automatically, NetBeans requires a lexer. A lexer is a class that translates the raw code input to tokens of a specific type, which can then be used by a parser to recognize their structure. This process will be explained in more detail in chapter 4. NetBeans only passes the actual content of the agent's program code to the lexer. However, the GOAL lexer requires the actual file as its input in order to resolve all dependencies properly. Thus, a system was put in place that created temporary files for the input that was to be lexed. In addition, more custom code was required to force NetBeans to re-color a file entirely at all times, as the default is to only re-color the currently edited line. However, due to the KR-sections (see sec. 1.1), this is impossible; one cannot know if we are in a KR-section at the current line or not.

At this early and initial prototypical stage of development, we did not modify the GOAL platform. Therefore, the GOAL lexer could not be used on its own, as required by NetBeans, because the GOAL parser decided which lexer to use at what time in order to use a different lexer for the KR sections. As different KR languages can be used within GOAL, the parsing infrastructure for these sections is separated from the GOAL constructs themselves. This process had to be imitated in the NetBeans plug-in by creating a new lexer that deferred its tasks to the specific KR lexer at the right moments itself. However, as a lexer does not know the structure of the tokens it recognizes, this had to be done on a low level by manually recognizing the keywords that initiate such a section, and then counting brackets in order to determine when to end the section. In addition, as the lexers partially contained the same type of tokens (e.g. single characters like brackets and arithmetic operators) but with different numeric identifiers, even more manual implementation was required in order to separate those numeric identifiers properly. However, after all of this was done, support for the basic editing of all related file types in NetBeans was created.

Advanced features like copying the indentation of the previous line when going to the next, folding blocks of code delimited by curly brackets, highlighting matching bracket pairs, and highlighting occurrences of the currently selected word in the rest of the code were quickly implemented by using default NetBeans features as well. However, showing possible syntactic or semantic errors and warnings to the user was more challenging. First of all, the GOAL core would only output those messages as plain text, requiring the plug-in to parse that text into the required objects by NetBeans manually. Moreover, NetBeans required a start and end position for

these messages that is based on the position of the related code in the file, starting from the beginning. However, as GOAL error messages just provide a line number and a character position on that line, this had to be translated into the character position relative to the entire document in the plug-in. External files like modules and knowledge bases were a problem within this structure. A single module or base can be used by multiple agents, but its semantic validity depends on the context provided by a specific agent. When editing such an external file, NetBeans requests the feedback for that file individually. However, the GOAL core did not support either semantic or syntactic validations of such files separately, as this was integrated in the feedback for a specific agent. To support this, more manual code was added that embedded the external module in an empty agent file in order to provide basic parsing. Moreover, an action was added that could use a specific agent file for this process, e.g. selecting a custom agent file instead of the default empty agent file, distilling the relevant feedback for the specific external file.

Auto-completion was added as well by storing the content of all tokens a lexer encountered in a set, whilst manually checking if the type of the token is relevant for the auto-completion (brackets are not, for example). The default tokens (e.g. keywords) present in the lexer were added to this set as well, together with the KR keywords (encoded manually within the plug-in for Prolog). This provided context-insensitive auto-completion for individual files that was refreshed completely on each edit of the relevant file.

Finally, support for running a system was added as well. In NetBeans, the run action is specific to an entire project by default. To create an implementation quickly, a GOAL project was restricted to having a single MAS file with the same name as the project, which would allow the file to be located easily based on the project. As a console is provided by default by NetBeans, the run action only started a new Java process that executed the command-line runner from the GOAL core. In a similar fashion, an action that launched the 'current' GOAL IDE was also provided, which would automatically run the specified file. Specifying breakpoints was made possible as well by implementing custom annotations within NetBeans. Implementing a custom annotation can be done within a few lines of code, but the same mapping problem from a character index to a line number and character position as mentioned before had to be solved here as well. When a project was debugged, all annotations in all files within the projects were serialized in a single string that was passed on to the GOAL core, to be decoded there into the relevant breakpoint objects again. Support for manually executing external SWI Prolog files was added as well by adding a field in NetBeans' preferences in order to obtain the path to a SWI Prolog executable, which could then be used in a similar fashion to creating a GOAL (Java) process.

Thus, in summary, the prototype allows a user to edit GOAL files within the NetBeans project structure, allowing a single MAS file within one project, but supporting all other file types (including SWI Prolog) as well. Advanced editing features like automatic indentation, code folding, bracket matching, occurrence highlighting, and

in-code errors/warnings were added as well. Finally, facilities for running a MAS through a command-line interface or the already existing GOAL IDE were added, including support for breakpoints. An example of the prototype's interface is shown in Fig. 2.1. The navigator is empty as support for a content outline was not added.

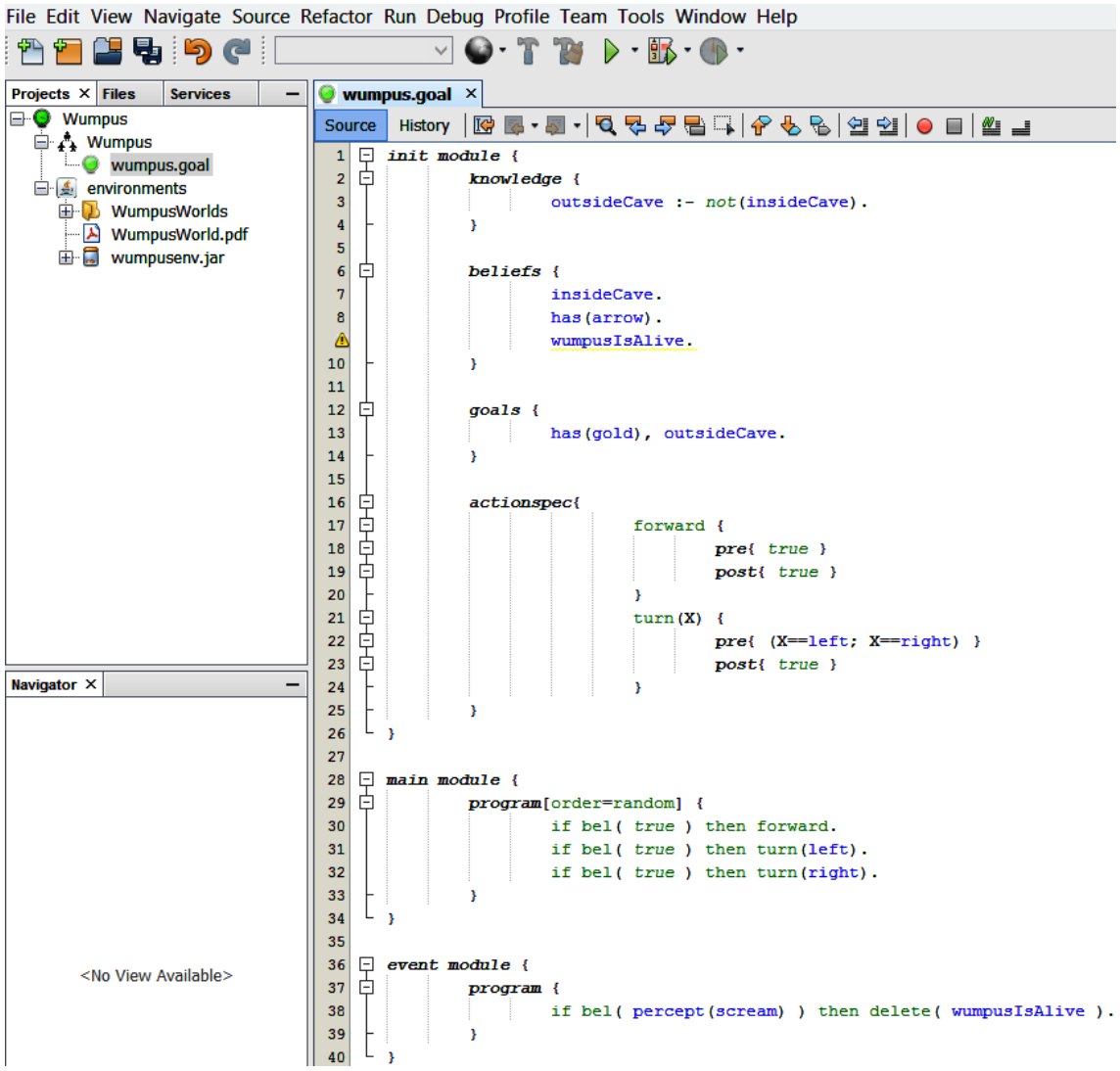

Figure 2.1.: The interface of the plug-in prototype created in NetBeans

Lessons learned The development of this prototype was a dynamic process, mainly tested by myself and a few fellow master students. First of all, it became clear that implementing such a plug-in in a maintainable way without changing any of the existing GOAL functionalities is not possible. Quite a few ad hoc solutions like the creation of temporary files for syntax highlighting, the manual switching between lexers, embedding module files, and linking the project to a single MAS file were used. This resulted in many assumptions about GOAL itself being hard-coded in the

plug-in, like which keywords indicate the start of a KR section. Thus, changes in the GOAL core are required for all of the mentioned issues. These required changes only became evident during the development of the prototype, which highlights one of the benefits of this effort. Thus, careful design of the plug-in itself and the resulting changes to GOAL is needed in order to create a reliable system.

Moreover, due to the design of the syntax highlighting and accompanying auto-completion, the editor's performance was not that good, especially for large files. On each edit, the event NetBeans generates for the current line was redirected to an event for the entire content, which had to be saved to a file first. On each of these events, all encountered tokens and default keywords were saved to a set for the auto-completion, and the plain text output from the GOAL core about the errors and warnings had to be parsed into objects, continuously translating string indexes based on the whole document into line numbers and column positions (or the other way around). As this process was too slow to be executed after each edit, it was continuously deferred by a third of a second between all edits, resulting in an update only when the user had not done anything for that amount of time, which could still possibly freeze the editor for a short amount of time at that moment. This does not result in a very convenient editing environment.

Another issue that quickly became evident was the large amount of windows that could be open when running a system. For instance, NetBeans, the GOAL IDE (runtime), and the interface for an environment could be running all together, resulting in tedious toggling between three large windows. In NetBeans, multiple files can be open as well, which together with multiple tabs per agent in the GOAL IDE result in a very large amount of graphical interfaces for a user.

However, some solid foundations were created, like the passing of breakpoints to the GOAL core. In addition, important lessons about the parsing structure of an IDE and its performance were learned, allowing a proper design to be made in the future. Moreover, it became clear that it is not very difficult to implement editing features in a platform like NetBeans. However, it is difficult to do this in a reliable and maintainable way.

3. Plug-in Development Framework

The two major IDEs for Java development are Eclipse and NetBeans. As GOAL itself is written in Java, these IDEs are logical options, as opposed to a Visual Studio plug-in for example. A quick prototype of the plug-in was developed with NetBeans as well, although most other agent platforms are based on Eclipse. Moreover, several scientific papers based on Eclipse and plug-ins for it can be found, whilst nearly none exist for NetBeans. Additionally, students use Eclipse in the start of their first year for the Java programming course, and other agent tools have plug-ins for Eclipse as well, which makes future integration with such tools more easy. Thus, based on experiences with NetBeans, the scientific foundation of Eclipse, Eclipse's usage within the university, and its use by other agent tools, the choice was made to use Eclipse as the platform for our plug-in. The internals of Eclipse will be discussed in the next section, after which a framework for our plug-in will be defined.

3.1. Foundation

The structure of the Eclipse platform is illustrated in Fig. 3.1. The platform is based on an open architecture in which each plug-in can focus on a specific area, adding new functionalities without impact to other tools. A common workbench is used to integrate the tools from the end user's point of view. Tools can be plugged into this workbench by using predefined hooks called extension points. The Eclipse platform itself defines some of these hooks, but each plug-in can define their own hooks for other plug-ins to use as well, creating a layered system of plug-ins. This structure provides users with a common way to work with the tools, whilst keeping developers away from integration issues. The use of these extension points is discovered dynamically through use of the OSGi framework: a widely used Java service platform that implements a complete and dynamic component model. This framework allows different Java components to be installed, started, stopped, and uninstalled without requiring a restart of the program, e.g. life cycle management.

The Eclipse workspace also defines a common way for managing resources like projects, files, and folders. Moreover, for building user interfaces, the workbench makes use of the SWT and JFace toolkits. SWT is a widget toolkit that forms an alternative to Java's AWT or Swing toolkits by using a system's native GUI libraries, creating a largely unique implementation for each platform. SWT was originally created by IBM, but is currently maintained by Eclipse, although it is used in several

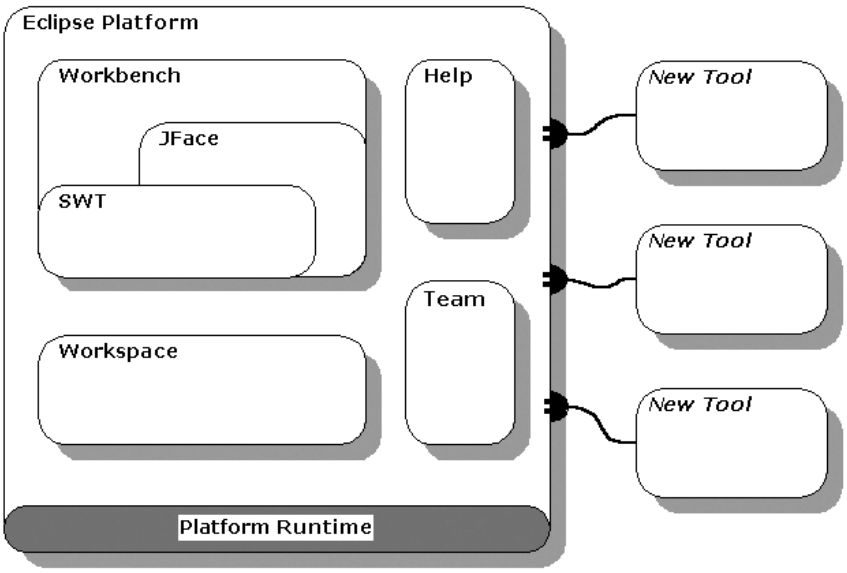

Figure 3.1.: An overview of the Eclipse Platform

other projects as well. JFace is a toolkit within the Eclipse project that provides helper classes for development with SWT based on the model-view-controller architecture. Finally, separate foundations for a help system, team support (managing and versioning resources), and debug support are embedded.

3.2. Language support framework

Several frameworks that provide support for implementing a language-supporting plug-in for Eclipse exist, all having their own advantages and disadvantages. These frameworks are all aimed at reducing the effort needed to implement such a plug-in by abstracting and thus standardizing parts of the Eclipse core. Four of these frameworks are currently available: IMP, Xtext, Spoofax, and DLTK. Based on our requirements, one of these will be selected, as the framework will need to support the implementation of all desired features. Moreover, proper developer documentation and example projects must be available, in order to actually reduce the required effort and increase the robustness of our implementation. First, these frameworks will be discussed shortly. Next, they will be compared with each other in order to select the framework that is to be used for our plug-in.

IMP[1] is a framework intended to "radically simplify the creation of an IDE". Many templates (‘meta tools’) are provided in which only a few customizations have to be

[1] http://www.eclipse.org/imp/

applied in order to get basic editing features working with any kind of parser. IMP has been around since early 2009, but the latest version was from early 2010; a lot of the mentioned features are not (fully) implemented yet. For example, only the LPG grammar language is actively supported, which is not a widely used language. There is nearly no documentation available. Moreover, at the time of writing this thesis, the project has even been archived, resulting in very few recent users of this framework.

Xtext[2] is a framework designed for domain-specific languages. Only a grammar in their own grammar language is required in order to create a full-fledged editing environment in Eclipse. This grammar is used to automatically generate ANTLR parsers and Java classes for the plug-in to use. ANTLR is a very powerful, versatile, and widely used parser generator. Xtext has been around since 2009, and is being actively developed. Xtext has quite a large amount of very specific languages that have been developed with it, usually Java extensions. Extensive documentation is available on their website.

Spoofax[3] is similar to Xtext, as it is also designed to create a full-fledged editor for domain-specific languages based on a grammar in their own grammar language. The Spoofax framework is based on the IMP framework, and developed at the TU Delft. Its first version was released at the end of 2011, and its latest version is from early 2013. Compared to other frameworks, the user base of Spoofax is small. A Wiki is available that contains a highlight of the features of Spoofax and some example projects.

DLTK[4] is somewhat similar to IMP, as this Dynamic Languages Toolkit platform is also an Eclipse framework intended to simplify extending the Eclipse IDE with new languages. However, instead of working with any parser, DLTK focuses on dynamic languages specifically, such as PHP, Perl, etcetera. DLTK has been around since 2007, and is still being actively developed and used. For instance, the de facto Eclipse PHP, Perl and Ruby on Rails plug-ins are all based on DLTK. A Wiki containing several tutorials is available, providing an example project based on supporting Python that makes use of ANTLR parsers as well.

Comparison Xtext and Spoofax are both designed for domain-specific languages. For both frameworks, a specific grammar language must be used, of which there are no examples for languages like Java or Prolog available. In addition, neither actively support embedded languages (the KR sections in GOAL files). Both frameworks,

[2] http://www.eclipse.org/Xtext/
[3] http://strategoxt.org/Spoofax/
[4] http://www.eclipse.org/dltk/

although Xtext especially, are focused on languages that are an abstraction of Java code, as many options to for example automatically compile with Java exist, and the examples are based on these kinds of languages as well. Finally, neither frameworks contain an explicit debugging infrastructure, although this is listed as a future feature for Spoofax, and Xtext contains support for breakpoints.

IMP is advertised as a framework for any kind of language. However, although not explicitly forcing a certain parser to be used, only the LPG grammar language is currently supported, which is not an industry standard, and there is no prospect of any future development. Moreover, the lack of documentation and example projects is an issue. Finally, IMP might be targeted at a too general set of programming languages, and not the framework that will yield the most efficient development process in our case. In other words, we might benefit from a framework that contains more abstraction or standardization than IMP does.

Finally, the DLTK framework is catered to dynamic programming languages. Similar to the idea behind IMP, no specific grammar language is required, allowing the generation of a full-fledged editor quickly. In addition, being catered to dynamic languages, process like compilation, interpretation, and even debugging are inherently supported as well. However, in order to fully use all of these features, a mapping of an existing grammar into a specific structure must be provided. It is not strictly required to use this structure, but features like the content outline and auto-completion depend on it. This project is around for the longest, and is still actively developed and used by some large projects. DLTK is also included by default in some Eclipse builds. In addition, DLTK provides a debugging framework that is based on the widely-used DBGP protocol.

An overview of the evaluation of each framework is presented in Tab. 3.1.

In conclusion, the DLTK framework seems the most suitable candidate for the development of a GOAL plug-in for Eclipse. It is catered to programming languages similar to GOAL, and is mature, widely used, actively developed, and properly documented. In addition, a large debugging framework is provided.

Framework	**IMP**	**Xtext**	**Spoofax**	**DLTK**
Target group	Any language	DSLs	DSLs	Dynamic languages
Public development	2009-2010	2009-current	2011-2013	2007-current
Grammar language	Any (LPG actively supported)	Xtext	SDF	Any (ANTLR examples)
Editor framework	Yes	Yes	Yes	Yes
Debugger framework	No	Basic (break-points)	No (in development)	Yes
User base	Average	Large (DSLs)	Small	Large
Documents	Poor (nearly none)	Very good	Good (no tutorials)	Very good

Table 3.1.: A comparison of the available language support frameworks

4. Grammars for Agent Programming

In this section, a short introduction of the process of converting a programing language into a form that is useable for execution is given. Next, the actual implementation that has been created will be presented. Finally, the error reporting is discussed.

4.1. Overview

When implementing a programming language, an application is required that can read the language and identify the elements in the language. Broadly speaking, a language is a set of sentences that are made up of phrases, that are in turn made up of phrases existing of either sub-phrases or vocabulary symbols. To react appropriately, an interpreter (an application computing or executing sentences) has to identify and differentiate between these components, which is generally called 'recognition'.

Programs that do this are called 'parsers' or 'syntax analyzers'. To specify the rules of a language, or the syntax, a grammar is required. A grammar is a set of rules expressing the language's structure. Parsing is usually broken down into two distinct stages. The first task consists of grouping characters into words or symbols (tokens), which is called 'lexical analysis', 'tokenizing', or simply 'lexing'. The tokens that are generated by a lexer are composed of a token type identifying the lexical structure and the matching text. These tokens are used for the second parsing stage, which mainly consists of building a 'syntax tree' (or 'parse tree') that records how the structure of the input was recognized. In such a tree, the leaves are always the input tokens, identified by phrase names in the elements above, increasing in the level of abstraction towards the root node. An example of such a tree is shown in Fig. 4.1 (for a *.mas2g* file in this case).

We use a parser that is based on 'recursive descent': a collection of recursive rules that starts at the root of a parse tree, proceeding towards the leaves (left to right). In general, this is also called 'top-down parsing'. Bottom-up parsing also exists, which involves locating the most basic elements, then the elements containing those elements, etcetera, which is also known as 'shift-reduce parsing'.

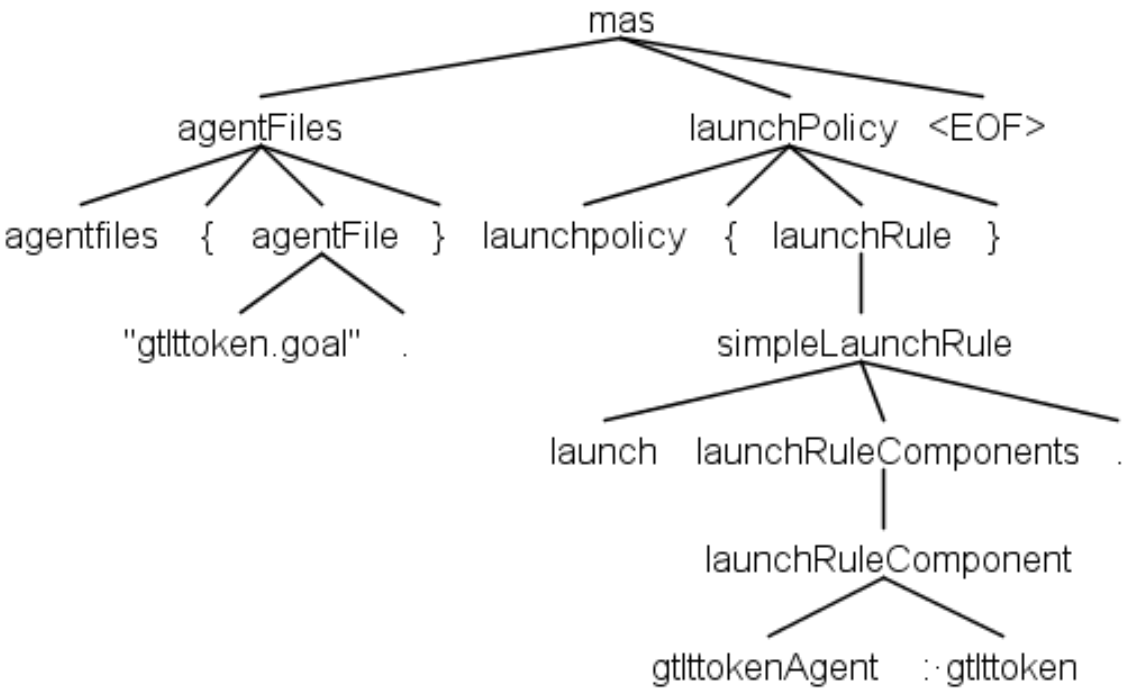

Figure 4.1.: Part of a parse tree

When facing alternatives in this process, 'parsing decisions' or even 'parsing predictions' will have to be made. This is typically done by using 'lookahead': looking at one or more subsequent tokens to decide which alternative will succeed, which is more efficient than simply backtracking at every mistake. However, ambiguities (like the famous 'you cannot put too much water into a nuclear reactor') can still cause problems. Syntax should be designed to be unambiguous, but when ambiguities occur usually the first valid alternative is chosen as the intended one, which relates to the derivation issues that exist in context-free grammars in general.

When a valid syntax tree has been built, it is up to the application to use it. For GOAL, for example, Java classes will have to be created that correspond with the given data, like the `IfThenRule` class (and its children). This usually involves 'walking' the tree, for which two main mechanisms exist. The first one is the 'listener pattern', in which the tree is visited in a depth-first pattern, firing an event that is to be handled by the program when visiting a node. However, when control over the walk itself is required, the 'visitor pattern' can be used. When using this pattern, instead of firing events, stepping to another node has to be done manually. This has the advantage of being able to use properties of children in the parent node (like multiplying two numbers in a multiplication for example).

Error reporting and recovery (e.g. dealing with ungrammatical sentences) is very important in all steps of this entire parsing process. Simple examples of this are single-token deletion: pretending an extraneous token is not there and single-token insertion: pretending a required but missing token is there. Even 'guessing' the

appropriate token type when a type could not be defined is possible. Much more advanced error recovery algorithms exist, but these will not be discussed here.

To simplify the implementation of languages, so called 'parser generators' exist. Parser generators are tools that create a parser (and optionally a lexer) from a grammar. Many of these tools exist, all having different lexing and parsing algorithms, input notations, output languages, categorizations, IDEs, licenses, etcetera. For GOAL, grammars were already written in ANTLRv3. Only recently, in January 2014, the next version of ANTLR has been released: ANTLRv4. ANTLRv4 introduces a new parsing technology (`ALL(*)`) that performs grammar analysis dynamically at runtime, which in practice decouples the grammars from the underlying parsing strategy. Tree construction and walking has been automated as well, decoupling the grammar itself from the processing of the eventual syntax tree, resulting in a more powerful and easier to learn platform that comes with a dedicated IDE (ANTLRWorks) and book[34]. Because of the existing grammars, experience with ANTLR, and the many improvements ANTLRv4 provides, it was quickly decided to use ANTLRv4 for the construction of the whole 'parser stack': a lexer, parser, and visitor interface in order to eventually use a custom-made visitor class that will translate source code into the Java objects that are used in the GOAL core.

4.2. Implementation

The GOAL language is constructed in such a way that it contains so called 'language islands': pieces of code (regions) in another (third-party) programming language that are embedded in sections of the GOAL code, which makes GOAL an 'islands language'. A GOAL file can theoretically use any valid knowledge representation language (KR-language), which is usually SWI-Prolog, in these specific sections. Another example of this is JavaDoc inside Java programs.

Modern IDEs like NetBeans and Eclipse, exclusively use a lexer (tokenizer) in order to do syntax highlighting. This is logical, as the (much more) expensive tree-building and tree-walking processes of a parser do not have to be performed to simply identify what kind of tokens are present. Lexing can be done very swiftly even when the user is still typing, whilst parsing might be done only when the user is finished (or at some predefined interval). Moreover, in the current GOAL IDE, a separate grammar (for JEdit) is used to facilitate the token highlighting, but such a duplicate solution is not maintainable; it was not uncommon that highlighting errors were present even though the parsing process went fine. Using the lexer itself ensures the correctness of the highlighting.

The decision of which tree walking method to use was relatively easy to make, as the way in which GOAL is structured is very inherent with the visitor pattern. For instance, 'listening' to the event of visiting a pre-condition would be useless without knowing to which action-specification object that condition actually belongs. This would be very hard in the listener-pattern, as in that case a

local `ActionSpecification` object would have to be stored for a child (e.g. a precondition) to 'add to' later on. By using a visitor pattern, these children can be 'retrieved' instantly by visiting the required children in the required order directly, thus avoiding the necessity of temporarily storing many 'parent objects' in the class.

In the GOAL ANTLRv3 grammars, there is a tight coupling between the GOAL and KR (Prolog) parser. The parser makes explicit decisions about which lexer to use in specific situations. However, this breaks the whole idea of separating this process into two independent steps. ANTLRv4 provides explicit support for language islands through lexical modes, which allow a lexer to switch between the different languages itself, as required for proper syntax highlighting. In addition, in the ANTLRv3 grammars, all the tree-building and tree-walking code was incorporated in the grammar definitions themselves, resulting in very illegible and highly coupled grammars. Thus, to improve the maintainability of the grammar, that code will have to be moved into the ANTLRv4 visitor structure.

Besides a grammar for the GOAL language itself, which includes *goal* and *mod2g* files, grammars for a MAS file (*mas2g*) and unit test file (*test2g*) exist. In the next part, the created grammars will be discussed briefly. However, this does not contain the grammar for the unit tests, as these were created in another project after the other grammars were finished; they were thus based on these grammars. The corresponding full ANTLRv4 grammars can be found in Appendix B.

MAS2G These files define a multi-agent system, and are composed of three sections:

- which environment to (possibly) use
- which agent (*.goal*) files to use
- when to launch agents and optionally connect them to the environment.

Redesigning this grammar was straight forward, as the original ANTLRv3 grammar did not need to be changed significantly. However, as mentioned before, the application specific code needed to be removed from that grammar. When all lexer and parser rules were recreated in a single *g4* file, ANTLRv4 was used to generate Java classes for both the lexer and the parser. Moreover, as mentioned in the previous section, a visitor interface was created. Implementing this visitor is also a very important step, as the actual translation to useable objects by the GOAL core and the corresponding error handling has to be implemented there. In such a visitor, a function can (or has to) be implemented for each of the parser rules. Traversing these functions, eventually, the actual lexer tokens will be reached, 'closing' that branch of the tree. Creating the visitor is a tedious process requiring much understanding of both the grammar itself and the definitions and requirements of the Java objects that need to be created, whilst also having to take possible errors made by the user into account. However, the split of lexing, parsing, and object creation into three different steps helps a lot in successfully completing this process.

GOAL The process for *.goal* files is nearly identical, with the exception of the aforementioned 'KR-sections' (language islands). For this, a special feature of ANTRLv4 was put to use: lexing modes. These modes makes sure that within the limits of the initial left and matching final right curly brackets, indicating these embedded sections, all contents are matched to a single collection of characters by entering a special lexing mode. This needed some extra Java code though, because unlike other island languages, the curly brackets may also occur within the block itself. Thus, the curly brackets are counted in order to determine when a closing bracket matching the opening bracket is encountered, after which the special lexing mode is dropped. Similar rules are applied for statements that use KR sections within regular brackets. Of course, when a bracket is missing, the lexer mode will not be correct at a certain point. However, in all circumstances, a lexer encountering tokens it does not recognize will generate recognition errors, allowing the user to fix the problem. In the corresponding visitor for GOAL files, these sections are put into a stream that is passed onto any corresponding parser that is registered for the KR language in GOAL, which is properly arranged through Java interfaces, allowing any KR implementation to be supported easily.

Due to the experience in creating the grammar and visitor for MAS2G files, implementing these for GOAL files, although having a much larger grammar, was doable after dealing with the challenges because of the language islands. However, a backwards incompatible change had to be made anyway, as the syntax of the `send` predicate was incompatible with the new 'bracket-matching process'. In the old syntax, a send-action could look like this: `send(allother,hello,me(agent))`. However, the 'all other' indicator is part of the GOAL grammar, and not a KR-specific predicate, but is within the brackets of the send predicate. In the new grammar, those brackets directly correspond to the KR section, forcing a change to: `allother.send(hello,me(agent))`. Although creating a backward incompatibility, this change was not completely undesirable, as a similar syntax already existed for working with mental models of other agents (e.g. `agent.bel(...)`), creating more consistency in the grammar as a whole.

KR As mentioned before, GOAL files have KR sections that contain code in another programming language like SWI Prolog. GOAL even uses a customized subset of SWI Prolog for compatibility reasons, written in ANTLRv3. This complicated grammar was not ported to ANTLRv4 for several reasons. First of all, an external plug-in that is based on DLTK as well, ProDT[1], is installed together with our plug-in by default. The syntax coloring of this plug-in is used within our own plug-in, but the GOAL parser uses the mentioned ANTLRv3 grammar, and is thus also responsible for generating any syntactic or semantic feedback. Moreover, ProDT provides convenient functionalities that allow a user to execute an (external) Prolog file for example. However, in order to not highlight any non-supported predicates,

[1] http://prodevtools.sourceforge.net/

and to not an extra dependency for a user to download as ProDT was based on a previous version of ProDT, a custom build of the plug-in was created. In general, support for any other KR language can be added quickly when an Eclipse plug-in for that language exists. The syntax coloring process is not specific to DLTK, and thus, the link of the syntax coloring with ProDT for example can be recreated for any other Eclipse-based language plug-in easily. Additional features like running an external KR file separately are not required, but do have an added value to the user when present of course, and the parsing itself is eventually still based on the GOAL parser. Another reason for not upgrading the Prolog grammar is the fact that it will not make use of any of the new features in ANTLRv4, like language islands, which the required effort not worthwhile.

Requirement: *A parsing structure for all file types (1)*
Requirement: *Dynamic support for embedded KR languages (2)*

4.3. Error reporting

When creating new grammars, first of all, one has to make sure not to break the compatibility with the existing specifications. To this end, a set of files known to be error-free was used in order to test the grammars. However, this approach also had a downside. For instance, unchecked nullpointers in the visitors could occur more easily than expected when making mistakes in to-be-parsed files. Handling all of these properly without making an editor crash is of course a necessity for a proper user experience. Other small issues, like only allowing a *.jar* extension for environment files whilst other extensions should be allowed as well, indicated that the current set of test files was insufficient. More examples of more different code files should be available, including files which deliberately contain errors.

The error messages given are also vital for allowing a user to identify the actual error. However, these messages can be generated at multiple levels. First of all, both the lexer and parser generated by ANTLR have their own error reporting. Next, the aforementioned visitors can cause exceptions that need to be handled as well. Finally, on a semantic instead of syntactic level, so-called validators exist in the GOAL core that check if a used predicate has been defined for example. In order to create a uniform system, the `WalkerInterface` was created. This interface in turn extends the `ANTLRErrorListener` interface. All created visitors implement this interface, which ensures that all errors and warnings can be obtained from two corresponding functions. These functions require a list of `ValidatorError` and `ValidatorWarning` objects to be returned, which are classes with general support for creating messages based on strings (error messages) that are defined in an external (text) file. In addition, the `WalkerHelper` class was created to aid in this process, resulting in the following example of a function call in the GOAL visitor:
`walkerhelper.report(new ValidatorError(GOALError.MODULE_MISSING_NAME,`

`walkerhelper.getPosition()))`. This piece of code reports an error about a module not having a name, together with the corresponding position in the source code. The `GOALError` constants correspond with a defined error message, as do `GOALWarning` constants. The same message classes are used within the validators, for which an abstract `Validator` class similar to the `WalkerInterface` exists, ensuring a consistent system of error and warning messages, of which the content has all been defined in one place. Currently, only the messages from ANTLR are taken as-is, as customizing them would involve a large amount of work, whilst those error messages are already well structured and informative.

5. Editing Framework

In this chapter, using the Eclipse plug-in framework and DLTK, the implementation of the skeleton for the GOAL plug-in itself and the editor will be discussed. Together with the next chapter, which discusses running and debugging a multi-agent system: the core of the implementation. The parts are ordered in such a way that one could follow these steps in order to create an Eclipse plug-in for another (agent) programming language. This is also one of the main contributions of these chapters; although some small and mostly object-oriented sources are available online, there is very little precedence for creating such a full-fledged plug-in. Online sources like the DLTK wiki[1], IBM developer works[2], and a few other websites exist, but are outdated, incomplete, or not focused on supporting a programming language. There is especially little documentation available for constructing a debugger. Existing open-source projects using DLTK do exist, ProDT[3] and Freemarker[4] for example, but these large code bases are not well documented by either external or in-code documentation, requiring a reverse engineering effort, the possible differences between requirements aside. Here, we try to discuss all aspects of creating a full-fledged IDE completely, using small parts of the mentioned sources and experience by trial-and-error in order to set the standard for future IDE developers. Thus, these chapters can be read as an implementation guide. And although focused on GOAL, nearly everything is generally applicable, especially for other agent or logic-based programming languages.

In Fig. 5.1, the general structure of an IDE is illustrated[12]. In this chapter, the implementation of the 'left half' of the development environment will be discussed: editing files in a structured way with optional customization and other user interface tools. In the next chapter, the 'right half' of the development environment and the accompanying adaption part of the run-time environment will be discussed. The existing GOAL core forms the run-time application.

5.1. Registering the plug-in

In this section, the first step is to create the plug-in itself will be discussed. First, we have to make sure Eclipse recognizes our plug-in, and loads the necessary com-

[1] http://wiki.eclipse.org/DLTK
[2] http://www.ibm.com/developerworks/library/os-ecplug
[3] http://prodevtools.sourceforge.net/
[4] https://github.com/angelozerr/Freemarker-Eclipse-DLTK

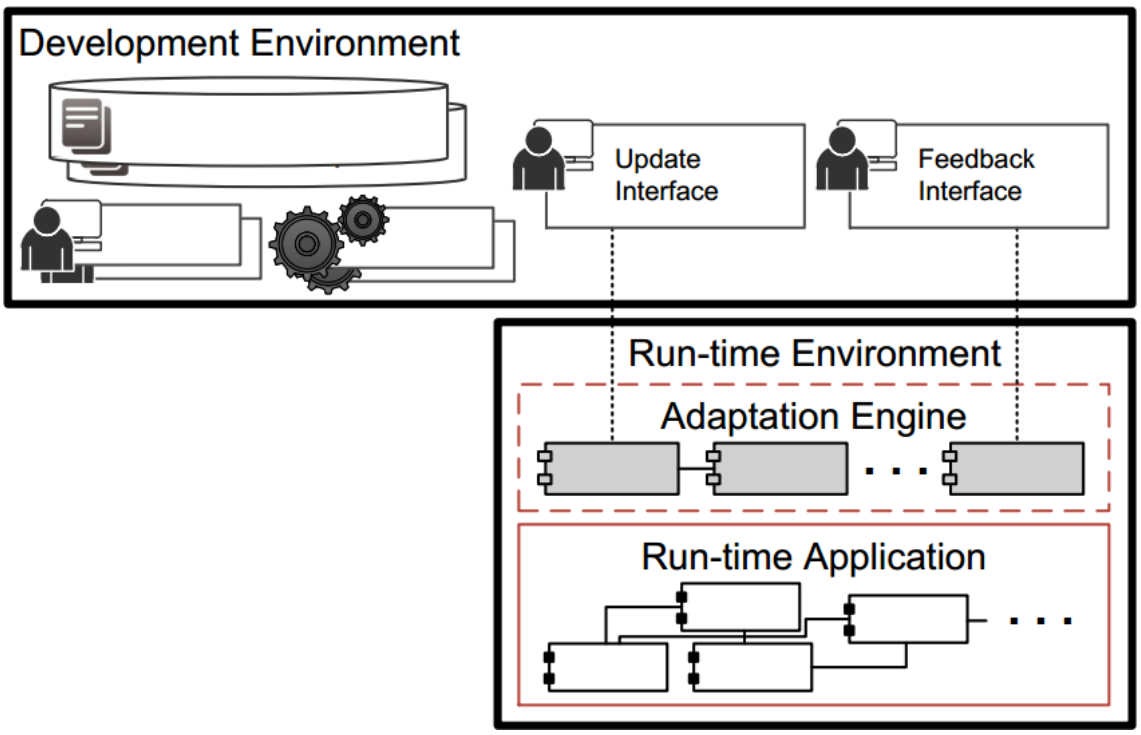

Figure 5.1.: The general structure of an IDE

ponents. Next, we ensure that files of types relevant to GOAL are processed by our plug-in, and allow a user to create new files as well.

Nature An Eclipse plug-in is loaded through a single *XML* file: the manifest. This file contains definitions of so called extension points, which are links to the implementation of specific functionality. All extension points Eclipse defines and their corresponding elements are available in a central database online[5]. The first extension point any Eclipse plug-in must define is `org.eclipse.core.resources.natures`, which defines the so called nature of the plug-in. A nature uniquely identifies a certain plug-in, and is often used to bind different kinds of functionalities within one plug-in. For example, a default icon-image can be provided through the `org.eclipse.ui.ide.projectNatureImages` by giving the path to the image and the nature to link to. More importantly, projects within Eclipse are assigned a certain nature, specifying the plug-in that should handle everything within the project. A nature is actually a Java class: `GoalNature`. This class allows DLTK to hook directly into the project. Besides this class, there is also the `Activator` class, which acts as the main class of the plug-in, being initialized when the plug-in is loaded by Eclipse. This class automatically allows a plug-in to save its own preferences and log files, and provides different functionalities such as logging and string externalization.

[5] http://help.eclipse.org/kepler/nav/2_1

Update center After giving the plug-in a unique identifier, the user needs to be able to install and update the plug-in. For this, first, a so-called 'feature' must be created: a collection of plug-ins. This is needed because functionalities can be spread over different plug-ins; a feature combines these into a single package for an end-user to use. Again, this is done by using a single *XML* file. Such a feature can then be integrated into an 'update site', which is again a single *XML* file. This file has to be available somewhere on the internet, and with the link to that file, the user will be able to download the feature, and thus the related plug-in(s). Because the site's *XML* also contains a version number for a feature, Eclipse will also be able to detect when the feature has been updated. The version number is updated automatically when building the update site, a process that also makes sure the required files are available (compressed into *JAR* files). For GOAL, the part of the SVN repository on which the update site's project resides has been made public, allowing updates to be pushed by simply doing an SVN commit.

Content type As we want to provide a plug-in that is capable of editing files, the next step to take is linking certain file extensions to the plug-in. This can be done using the `org.eclipse.core.contenttype.contentTypes` extension point. It should be noted that only one extension point of the same type can be provided per plug-in; we cannot split the different extensions (i.e. *mas2g* and *goal*) into different extension points and thus different IDs. Optionally, a content describer can be used in order to match files that do not have a certain extension to the plug-in anyway by their content. For GOAL files, however, this is difficult, as a file can have a lot of different formats; no convenient header (as with Python) or anything exists. In addition, the required effort of implementing such a feature is not in balance with respect to the size of the problem; files without extensions are rarely used, and giving them a valid extension is simple. Thus, for GOAL files, this feature was not implemented.

Language toolkit After these initializations of the Eclipse plug-in basics, we move on to the initialization of some of the DLTK basics. The first required extension point is `org.eclipse.dltk.core.language`, which specifies a so called language toolkit to use: the `GoalLanguageToolkit`. This class links the plug-in's nature to the specified content-type extension point, identifying files of the given content-type automatically and allowing our plug-in to handle them. Similarly, a toolkit for the UI is specified through the `org.eclipse.dltk.ui.language` extension point and the `GoalUILanguageToolkit` class. This class contains a link for the UI to the previously specified `GoalLanguageToolkit`, ensuring that files identified by the toolkit are shown in an editor that is managed by DLTK, allowing other parts of the plug-ins to easily customize the appearance.

Wizards and files The last basic initialization is that of a project wizard, able to create projects for users to work with. Multiple wizards can be registered through the `org.eclipse.ui.newWizards` extension point. For these wizards, four abstract classes were defined: `NewGoalProject(File)Wizard` and `NewGoalProject(File)WizardPage`. The `NewElementWizard` classes are the main entry points for registration in the extension point. They can contain multiple pages, and provide the actual creation of a certain element (e.g. finishing the wizard). In our case, only a single page per wizard was used. These pages provide the actual layout of the wizards by using elements from either JFace (the Eclipse UI elements) or the DLTK UI core (based on JFace). DLTK provides convenient elements like `StringDialogField` and `ComboDialogField`. The titles and optional texts in the dialogs are abstracted in a so called message bundle using the Eclipse NLS framework, in order to keep all custom user interface text in a central file (*messages.properties*). All wizard pages contain their own validators, based on an embedded abstract class in our abstract class, which checks the validity of the input provided by a user in each input field. For example, the new project wizard checks if the name provided for the project is not empty and does not yet exist.

The 'Existing GOAL Project' wizard is defined through the `org.eclipse.ui.importWizards`, and uses the same classes as the previously mentioned wizards. However, this wizard is special, as it is listed under the import option of Eclipse. This is because it facilitates importing existing GOAL projects into Eclipse by selecting the corresponding *mas2g* file. The selected file is parsed, allowing that file and all referenced (sub)files to be used in or copied to the workspace.

Fig. 5.2 shows a short overview of the available wizards for the GOAL plug-in.

Requirement: *Managing projects composed of the supported file types (3)*
Requirement: *An importer for existing (old) projects (7)*

Templates Templates are an additional functionality of the wizards for new files. When creating a new file, some default code can be provided within this file. This works through the `org.eclipse.ui.editors.templates` extension point, which needs a so-called template context. This is provided through the `GoalUniversalTemplateContextType` class, which contains a unique identifier for the templates within our plug-in and a factory method to create the actual `GoalTemplateContexts`. All templates are provided within an extension point in the plug-in's manifest, linking to our context's identifier and providing some default content. These templates can then be used by our wizards by using the `GoalTemplateAccess` class, which provides an interface to the linked template context. When creating a new file, a template with a certain identifier is requested and used to fill the contents of that file by default. It is possible for users to edit these templates via the preference menu, which is described later in this chapter.

Of course, a design is required for these templates. Ideally, all elements provided in the template should actually be used in the eventually complete file. Otherwise, a

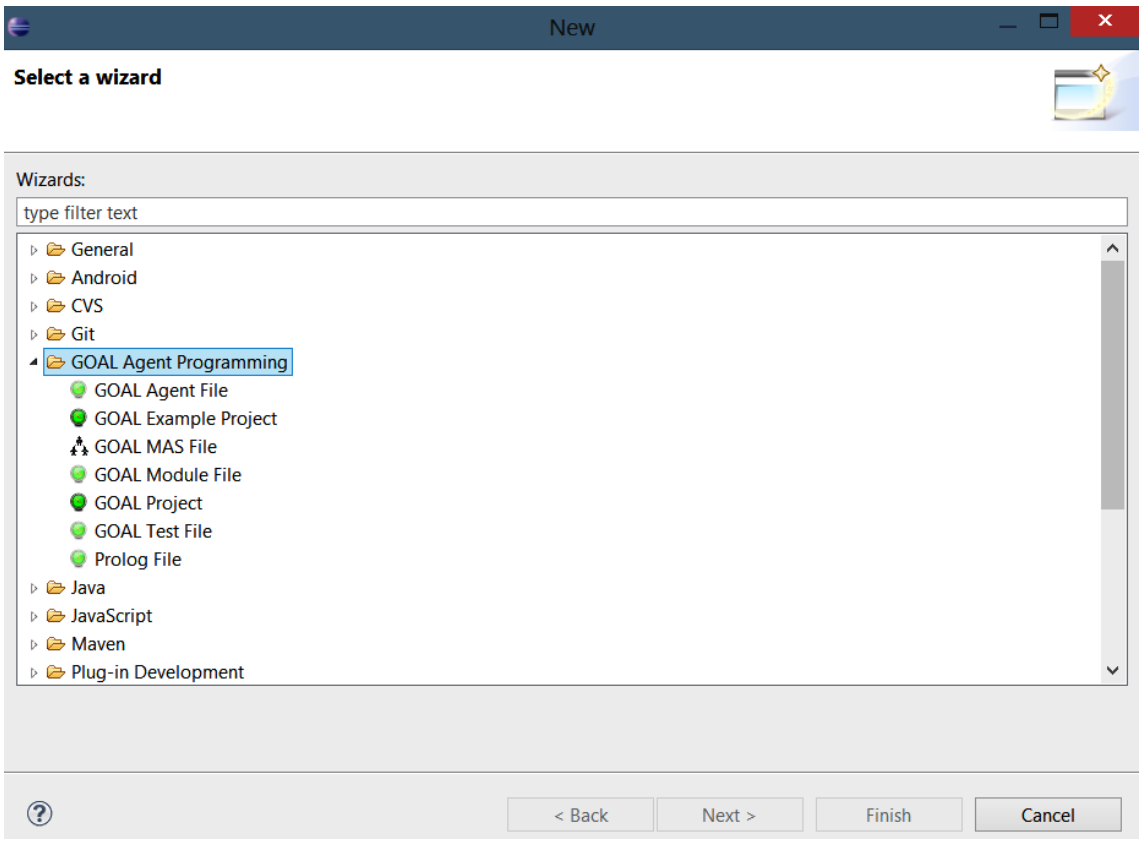

Figure 5.2.: The 'New' dialog in Eclipse showing the GOAL Agent Programming category

user is forced to delete parts of a new file every time. Moreover, a new file should not cause any errors, potentially confusing a new user. This might even need changes in other parts, like the grammar. For example, the template for a MAS file consists of an empty `agentfiles` and an empty `launchpolicy` section. Both are needed to run a MAS, as one always needs to register an agent file and specify when that agent needs to be launched. The optional environment section is not included by default; a separate template that does include that section is used when an environment is indicated in the dialog for a new MAS file. However, in order to not immediately generate errors, `agentfiles` and `launchpolicies` sections are allowed to be empty by the parser. Moreover, creating a launch policy without an according agent file or specifying an agent file without an according launch policy does generate an error or warning. This ensures a proper work flow for both beginning and experienced users, not immediately scaring them off with errors or large (possibly unnecessary) code sections, but preventing mistakes as well.

Requirement: *Increased programming support through file templates ... (8 partially)*

Class overview An overview of the classes that were discussed in this section is given in Tab. 5.1. The main package for the classes in this and all other overviews is `org.eclipse.gdt`. In the following section, the user interface of the plug-in will be discussed.

Class	Package	Extends/Implements
GoalNature		org.eclipse.dltk.core.ScriptNature
Activator		org.eclipse.ui.plugin. AbstractUIPlugin
GoalLanguageToolkit		org.eclipse.dltk.core. AbstractLanguageToolkit
GoalUILanguageToolkit	ui	org.eclipse.dltk.ui. AbstractDLTKUILanguageToolkit
GoalWizardPage	ui.wizard	org.eclipse.jface.wizard. WizardPage
NewProjectWizard	ui.wizard	org.eclipse.dltk.ui.wizards. NewElementWizard
NewProjectWizardPage	ui.wizard	GoalWizardPage
ExampleProjectWizard	ui.wizard	org.eclipse.dltk.ui.wizards. NewElementWizard
ExampleProjectWizardPage	ui.wizard	GoalWizardPage
ImportProjectWizard	ui.wizard	org.eclipse.dltk.ui.wizards. NewElementWizard
ImportProjectWizardPage	ui.wizard	GoalWizardPage
NewGoalProjectFileWizard	ui.wizard	org.eclipse.dltk.ui.wizards. NewElementWizard
NewGoalProjectFileWizardPage	ui.wizard	GoalWizardPage
NewAgentFileWizard	ui.wizard	NewGoalProjectFileWizard
NewMASFileWizard	ui.wizard	NewGoalProjectFileWizard
NewModuleFileWizard	ui.wizard	NewGoalProjectFileWizard
NewPrologFileWizard	ui.wizard	NewGoalProjectFileWizard
NewTestFileWizard	ui.wizard	NewGoalProjectFileWizard
GoalUniversalTemplate ContextType	completion	org.eclipse.dltk.ui.templates. ScriptTemplateContextType
GoalTemplateContext	completion	org.eclipse.dltk.ui.templates. ScriptTemplateContext
GoalTemplateAccess	completion	org.eclipse.dltk.ui.templates. ScriptTemplateAccess
Messages		org.eclipse.osgi.util.NLS

Table 5.1.: An overview of all classes discussed in this section

5.2. The interface

The interface of the Eclipse development environment is based on so called perspectives. A perspective defines a group of visual elements and their corresponding positions and sizes in the Eclipse window. Perspectives also define the menus: which actions are available in what category. In general, a perspective includes a *Package Explorer*, *Task List*, *Code Outline*, *Problems Overview*, and a large space in the center of the screen to edit files in. Moreover, for Java for example, the *File > New* menu contains options like *Class*, *Interface*, *Enum*, etcetera. There is also a separate debug perspective that includes different windows and menus. All of these perspectives are user-customizable.

Our plug-in defines a GOAL perspective, which will be discussed in the remnants of this section.

GOAL perspective The basics of the GOAL perspective, its name and icon, are defined through the `org.eclipse.ui.perspectives` extension point. One could use the corresponding `GoalPerspectiveFactory` class for building the perspective in code, but through the `org.eclipse.ui.perspectiveExtensions` extension point, this can more easily be done in the manifest itself. In here, the elements like `newWizardShortcut` (opening the aforementioned wizards for new projects/files) and `views` (content outline, console, etcetera) and their positions can be defined. As a side note, the wizards for GOAL projects/files also include a reference to our perspective, as the perspective will then be opened automatically after creating a GOAL project for example. Users can also manually open the perspective first, in order to get easy access to all of the GOAL project and file options.

The lay-out of this perspective requires careful design. Currently, it does not deviate much from other widely used perspectives like the Java perspective, or from that of other agent-programming plug-ins based on Eclipse. Fig. 5.3 shows what the current GOAL perspective looks like.

5.3. Editing files

Nearly all features discussed above were implemented through the plug-in's manifest or classes that extend some framework class and/or link other classes together. However, editing all possible GOAL file types need to be supported by the plug-in, and this requires more customized code, which will be discussed in this section.

Abstract Syntax Tree In order for many DLTK specific features to work, an Abstract Syntax Tree (AST) is needed. The ANTLR parser can also generate ASTs. The ASTs that DLTK expects are different from the ASTs ANTLR generates, however, and a translation is needed from the ANTLR nodes (classes) to DLTK nodes

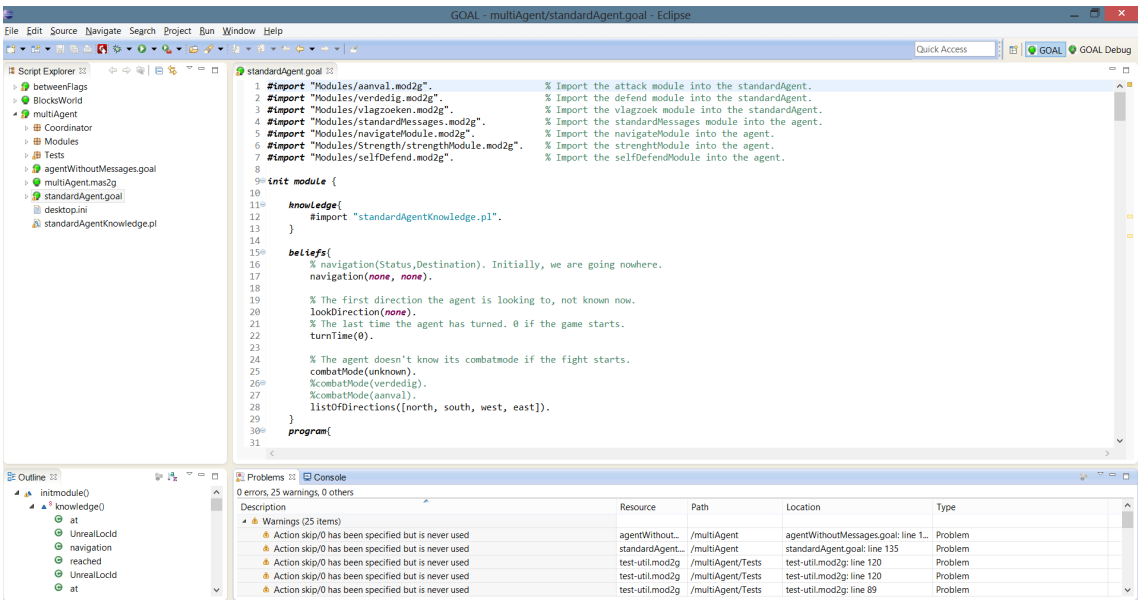

Figure 5.3.: The GOAL perspective

(classes). DLTK requires a tree that uses a fixed set of generic, higher-level nodes, found in the `org.eclipse.dltk.ast` packages. DLTK distinguishes the following node types:

- Declarations: arguments, field declarations, method declarations, type declarations, and module declarations (used as the top-level node for a source file).
- Literals: numerics, booleans, call arguments, call expressions, and strings.
- References: constants, types, and variables; holders of literals.
- Statements: a single statement or a block of statements.

These nodes only need to know their type, position, length, and children in order to be used properly. All of these classes require the specification of their position and name, and can have specific child nodes depending on their own type. It is also possible to set some properties on these nodes, like a method being static, a variable being a constant, etcetera. The ANTLR visitor pattern discussed earlier allows creating a tree like this, and therefore this mechanism was used, and a new visitor class was created, `GoalSourceWalker`, which visits all nodes defined in the grammar in the same way as the `GoalWalker` in the GOAL core. However, instead of creating objects from the GOAL core, classes from the DLTK AST package are created. For example, a single GOAL file starts with a `ModuleDeclaration` as its main node. For all modules in the file, a `MethodDeclaration` is added to that `ModuleDeclaration`. As modules can have arguments, these might be added to them using the `Argument` class from DLTK. This mapping process is applied to all other nodes of a GOAL file as well, and visitors have been provided for MAS and test files. Using this pattern

is good for the maintainability of these classes; the GOAL core itself does not need to take any DLTK aspects into account, but changes in any of the grammars will require the related visitor to be updated as well.

It is obvious that a mapping problem might exist here, as one might for example argue that a module is not a method. For *.goal* files, the mapping looks like this:

- `MethodDeclaration`: fixed language constructs like modules and their sections (e.g. goals, beliefs, etcetera).
- `TypeDeclaration`: single rules (e.g. if-then, forall-do, etcetera) and their subparts.

Specific literals or references are also used for names and arguments. Although this mapping is quite basic, it is sufficient for the purposes of the DLTK AST: providing an internal structure to represent syntax information in such a way that it can interface to convenient but non-vital features such as the source outline, search, documentation, auto-completion, etcetera. These features do not need very detailed information, but they do need to know the rough structure of the file, e.g. where sections of a general type are. However, mainly because of the inherent bias to object-oriented or imperative languages, many of the discussed node types remain unused in this case. Prolog, for example, consists of atoms, variables, and terms, which do not properly fit into this model.

The content outline for a file is an example of how a DLTK AST makes the implementation of certain features much easier. To create such an outline, the `org.eclipse.dltk.core.sourceElementParsers` extension point can be used, linking to the `GoalSourceElementParser` class. This class, in turn, only links our plug-in's nature and the `GoalSourceElementRequestor` class, which only links to the already created tree to provide all necessary functionality.

Parsing With a valid DLTK AST, by using the `org.eclipse.dltk.core.sourceParsers` extension point, a 'source parser factory' can be defined for our nature: the `GoalSourceParserFactory`. This factory creates parsers for the file types we have defined for our nature earlier, which are instances of the `GoalSourceParser` class. Only one function has to be implemented in this class: parse a source, and return the `ModuleDeclaration` for it (e.g. the root node of a DLTK AST). First of all, because a plug-in can only support one set of file type, the function looks up the file that has to be parsed, and checks its extension. For any supported extension, *.goal* for example, the corresponding lexer and parser from the GOAL core are created first. Next, using the corresponding visitor, the `GOALProgram` is fetched, and then passed through the `GOALProgramValidator` from the GOAL core for semantic analysis. This is done in order to get any errors/warnings from the parser and/or the validator; they are shown in the files by using so called 'markers': objects that implement Eclipse's `org.eclipse.core.resources.IMarker`. These markers are annotations of a file: for a certain line number and character range

in that line a message (of a certain severity) can be provided. Eclipse will show these markers in the correct line in the editor, and underline the corresponding source code according to the severity (e.g. yellow for warning, red for error). Moreover, all markers from all files are gathered in the *Problems* tab; an Eclipse default overview of all markers in all files. After this error/warning gathering process, the `GoalSourceWalker` is used to provide the requested DLTK `ModuleDeclaration`, on which various functionalities such as auto-completion, the source outline, folding, and more are based. For *mas2g* and *test2g* files, this process is virtually identical.

Syntax highlighting A file is colored instantly, e.g. whilst the user is typing, while the parsing is not necessarily done directly. This is because parsing is a heavier process than just lexing the file; a large part of the interface depends on the fact that generating the presentation itself is lightweight and of low cost. In order to do this, a custom editor has to be defined in the plug-in's manifest through the `org.eclipse.ui.editors` extension point. This point links to our `GoalEditor` class. This class links the DLTK language toolkit to the `GoalTextTools` class. This class, in turn, of which one instance is managed within the plug-in's activator, links to the `GoalSourceViewerConfiguration` class. Finally, this class contains multiple settings for viewing a source, and is responsible for creating a `GoalSourceScanner`. An instantiation of this class is used in a DLTK `DefaultDamageRepairer`, which is in turn passed to a DLTK `(Script)PresentationReconciler`, which the `GoalSourceViewerConfiguration` requests for a certain source. To the `GoalSourceScanner`, a `ColorManager`, `PreferenceStore`, and `TextEditor` are passed. The `PreferenceStore` is a map of preferences that exists per plug-in (through the `Activator`). Initializing these preferences is done through the `org.eclipse.core.runtime.preferences` extension point, which links to the `GoalPreferenceInitializer` class. In this class, the colors for certain token types are initialized. These token types, in turn, are defined in the `IGoalColorConstants` class, that contains a set of '`public static final`' Strings like `GOAL_COMMENT`, `GOAL_STRING`, `GOAL_KEYWORD`, etcetera. In the `GoalPreferenceInitializer`, for each of these constants, a default value (using Eclipse's `RGB` class) is defined. Moreover, using suffixes to the named constants, preferences like 'should the section be bold or italic' are also defined there.

Back to the `GoalSourceScanner`: in this class, first, the `setRange` method is called. This passes a `Document` and an offset/length; a range in the document's content to do the syntax coloring for. However, because GOAL files contain embedded sections that use a different lexer, parsing only a certain range of text is not possible. Thus, in the `GoalSourceScanner`, the whole `Document` is passed into a new lexer, depending on the extension of the file that is to be lexed. Next, three other functions have to be defined that do the actual work: `nextToken` (returning an `IToken`: the next element in the document), `getTokenOffset` (the offset of the last read token), and `getTokenLength` (the length of the last read token). For GOAL files, a GOAL lexer is created in the `setRange` function. In the `nextToken` func-

tion, the next ANTLR token is requested from the lexer. In case this token is an embedded section (`KR_BLOCK` or `KR_STATEMENT`), that whole embedded section is passed to the `PrologCodeScanner` from the ProDT extension. Whilst End-Of-File (EOF) is not encountered by this embedded `CodeScanner`, calls to `nextToken` of the `GoalCodeScanner` are forwarded to the `PrologCodeScanner`. When a regular GOAL token is found, a new `IToken` is created, based on the lexed token from ANTLR. An `IToken` has functions like `isWhitespace`, `isEOF`, and most importantly: `getData`. This function requires a `TextAttribute`: a class from the Eclipse SWT core that indicates if that token should be of a certain color, bold, italic, etcetera. Next, the ANTLR token type is checked, and converted into an `IGoalColorConstant`. For such a constant, the `PreferenceStore` is consulted to get the correct settings. Here, the `ColorManager` also comes in to convert the `RGB` setting from the `Preferences` into a `Color` class that is required by Eclipse's interface. In this way, the syntax highlighting for GOAL files has been implemented in a reliable and lightweight manner. For *mas2g* and *test2g* files, the process is easier, only having to convert the ANTLR `CommonToken` into an Eclipse `IToken`, and thus not having to deal with embedded sections. Delimiting tokens such as brackets and quotes are automatically recognized by Eclipse, and thus the code editor supports the highlighting of pairs of such tokens by default.

This approach forces correspondence with the real grammar, and thus also the correctness of the highlighting, as only the actual lexers and their tokens are used, as opposed to for example implementing a separate mechanism for syntax highlighting, as was the case in the 'old' GOAL IDE. If a token is modified or removed, the scanner will have to be edited to work again, instead of silently failing. A new token will result in it not being highlighted, but this can easily be added when required. A small thing that is done outside of this structure is the separation of comments from actual code, which is required by some advanced DLTK features as discussed in the next section. This is done by using the `GoalPartitionScanner` together with the `IGoalPartitions` class (which contains the possible partitions). This class is linked to the source in the aforementioned `GoalTextTools` class. It requires a set of `IPredicateRules`: a set of rules that indicate blocks of code. In our case, these are: `EndOfLineRule` (on `%`) and `MultiLineRule` (on `/* .. */`), which indicate that from a percent character until the end of the line a comment is present, or everything between the block comment separators. These classes originate from Eclipse's JFace package, and allow us to implement the separation in a few lines of code. The comment characters are the same in all our file types, and are unlikely to change, so this additional feature should not cause any maintenance issues.

Requirement: *A code editor with proper syntax highlighting and error reporting (4)*

Requirement: *Increased programming support through ... bracket highlighting ... (8 partially)*

Requirement: *Improved support for program comprehension through a source outline ... (9 partially)*

Class overview An overview of the classes that were discussed in this section is given in Tab. 5.2. The next section will discuss additional user interface features that have been implemented in our plug-in.

Class	Package	Extends/Implements
GoalSourceWalker	parser	goal.parser.goal. GOALParserBaseVisitor
GoalSourceParserFactory	parser	org.eclipse.dltk.ast.parser. ISourceParserFactory
GoalSourceParser	parser	org.eclipse.dltk.ast.parser. AbstractSourceParser
GoalSourceElementParser	parser	org.eclipse.dltk.core. AbstractSourceElementParser
GoalSourceElementRequestor	parser	org.eclipse.dltk.compiler. SourceElementRequestVisitor
GoalEditor	editor	org.eclipse.dltk.internal.ui. editor.ScriptEditor
GoalTextTools	editor	org.eclipse.dltk.ui.text. ScriptTextTools
GoalSourceViewer Configuration	editor	org.eclipse.dltk.ui.text. ScriptSourceViewer Configuration
GoalSourceScanner	parser	org.eclipse.jface.text. rules.ITokenScanner
GoalPreferenceInitializer	prefs	org.eclipse.core. runtime.preferences. AbstractPreferenceInitializer
IGoalColorConstants	editor	
GoalPartitionScanner	editor	org.eclipse.jface.text.rules. RuleBasedPartitionScanner
IGoalPartitions	editor	

Table 5.2.: An overview of all classes discussed in this section

5.4. Extra features

By using a DLTK AST, convenient features can be provided in an easy way. All of these features, like auto-completion, code folding, and more will be discussed in this section.

Auto-completion Eclipse provides users the possibility (with *Ctrl-Space*) to complete or suggest pieces of code. To provide this, the

`org.eclipse.dltk.core.completionEngine` extension point from the DLTK core can be used, which specifies a certain completion engine for a nature; the `GoalCompletionEngine` in our case. This class first uses the corresponding lexer to load all known keywords into a set. The only function to implement is `complete`, which passes a DLTK `IModuleSource` and a position and offset, for which the local `createProposal` function can be called any amount of times in order to generate the suggestions for the user. Such a proposal is a class of the type `CompletionProposal`, which contains information on how to display and/or insert the completion itself. In the `createProposal` function, the first thing determined is the 'prefix': a possible set of characters the user wants us to complete. Next, the whole current document is traversed (using its AST) in order to fetch all other user-created elements.

Some other classes are needed to complete this system for DLTK. First, the `org.eclipse.dltk.ui.scriptCompletionProposalComputer` extension point must be used to the `GoalCompletionProposalComputer`. This uses some other utility classes: `GoalCompletionProposalCollector`, `GoalCompletionProposal`, and `GoalOverrideCompletionProposal`. Finally, the `GoalSourceViewerConfiguration` was edited in order to link to the `GoalCompletionProcessor`; a class that extends override Eclipse's default auto-completion functionality with that of DLTK.

Another type of auto-completion is automatically inserting characters, like an asterisk in the next line of a comment block, completing a quote, or even completing blocks of code (with regular or curly brackets). This can be done by providing an 'auto-edit strategy' to the `GoalSourceViewerConfiguration`: the `GoalAutoEditStrategy`. This class also deals with automatic indentation of code, e.g. setting the amount of tabs correctly after a newline, which is very comfortable for the user. This quite complicated class deals with many situations: should something happen after a bracket is typed, or a quote, or a newline, etcetera, and what and where should we insert something automatically. All of this can be fine-tuned by a user through the preferences, which are also registered in the `GoalSourceViewerConfiguration` by linking to the `GoalContentAssistPreference` class. Fig. 5.4 shows an example of how the auto-completion might look to a user.

Requirement: *Increased programming support through ... auto-completion, automatic indentation ... (8 partially)*

Documentation When giving suggestions to the user on how to complete a certain piece of code, it is useful to display information with each of the suggestions. For user-created elements, this can be done by using the user's own code comments (if any). DLTK has built-in functionality to transfer code comments into documentation, which besides from showing in auto-completion also works when hovering certain pieces of code. In order to implement this, the `org.eclipse.dltk.ui.scriptDocumentationProviders` extension point is used to link to the `GoalDocumentationProvider` class. In this class, the only function we

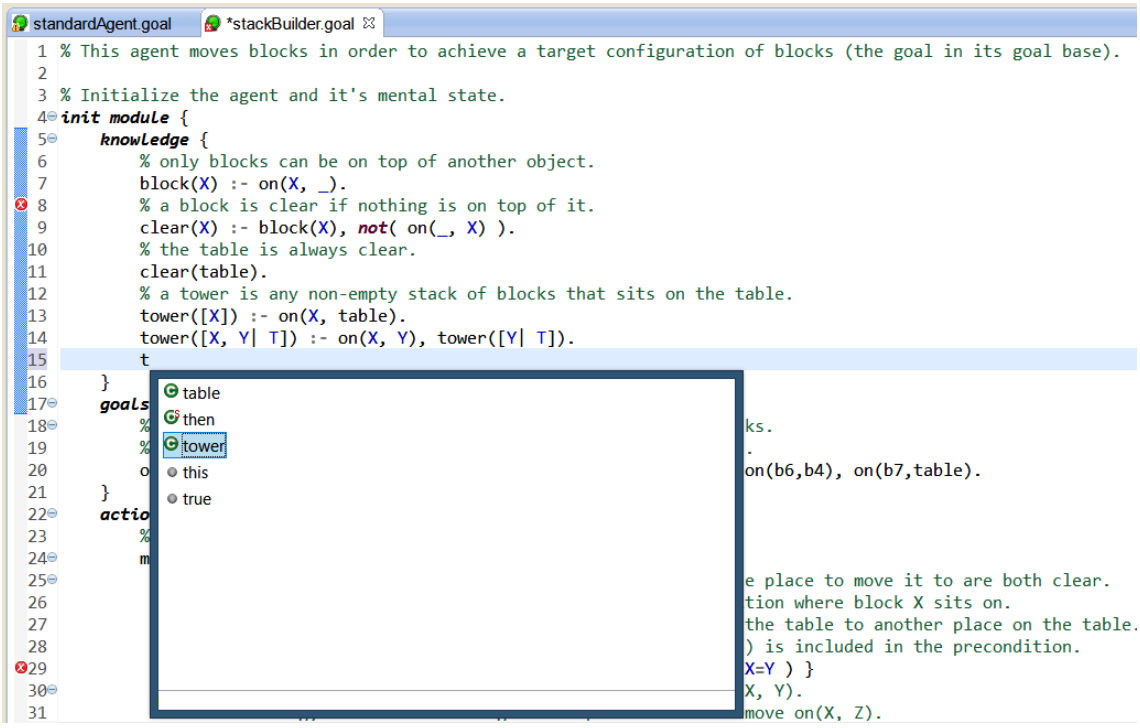

Figure 5.4.: An example of auto-completion on *Ctlr-Space*

need to implement is `getInfo`, which expects a HTML document for a certain AST element. For every occurrence of the given AST element, the line above is checked for a comment. If there is any, it is added to a set, which is then translated into a set of HTML-formatted suggestions. Fig. 5.5 shows an example of how the documentation might look to a user.

However, mainly because there is no fixed structure for API comments in GOAL, like e.g. JavaDoc for Java, this documentation can get cluttered. If, for example, multiple predicates are used on one line, and a single comment is written above that line, the comment will be applied to all of those predicates, whilst that might not be desired. Designing such a commenting structure that generates proper documentation that is perhaps even fit for exporting is out of the scope of this thesis, but would certainly increase the usability significantly.

Requirement: *Improved support for program comprehension through ... and the generation of API documentation (9 partially)*

Folding Collapsing certain sections of code or blocks of comments can be useful at times. It allows the user to focus on the parts of code that are relevant at that moment. DLTK provides functionalities to do this, again using the AST, through the `org.eclipse.dltk.ui.folding` extension point. Here, `blockProviders` and `structureProviders` can be set in order to determine what pieces of codes are

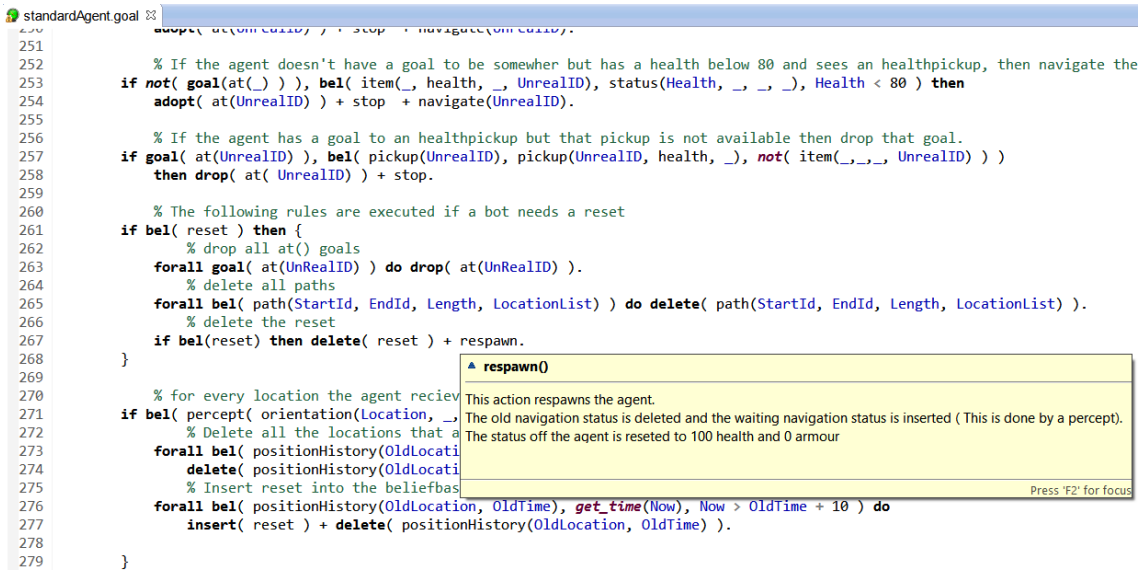

Figure 5.5.: An example of code documentation when hovering a predicate

blocks that can be folded. In our case, the `blockProviders` are `GoalFoldingBlockProvider` and `GoalFoldingCommentProvider`. The block-comment folding is based on the aforementioned code partitioning, and indicates that blocks of comments found by the aforementioned `GoalPartitionScanner` can be folded entirely. Multiple lines of single-line comments are automatically merged. The actual code-block folding uses a `ModelElementVisitor` to traverse the AST, and based on the type of a node that is encountered, it is determined if that node can be folded or not. This is mainly done on pieces of code that are real blocks, usually indicated by a `{ ... }` delimitation. It is useful to note that creating a DLTK-based AST has saved us a lot of trouble on many subjects already. An example of how folded code can look like in the GOAL editor is shown in Fig. 5.6.

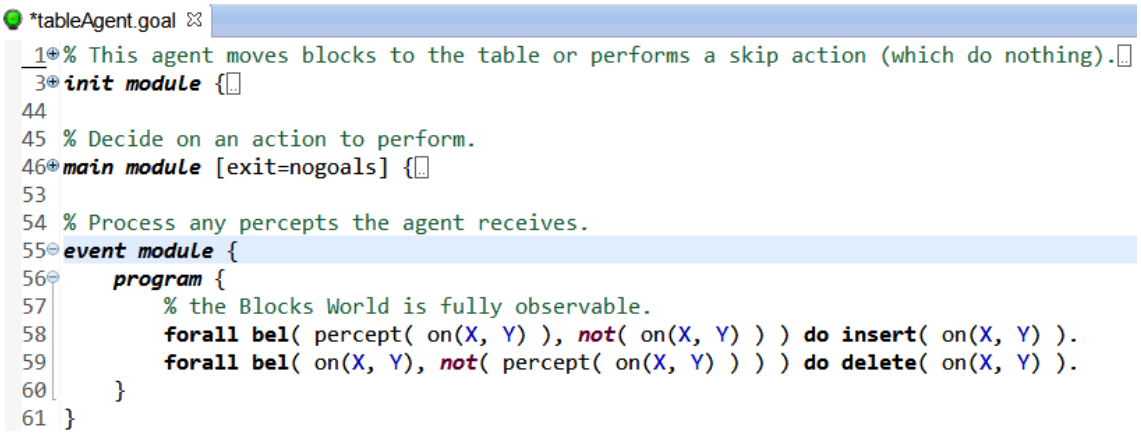

Figure 5.6.: An example of code folding

Requirement: *Increased programming support through ... and code folding (8 partially)*

Preferences Some parts of the plug-in are customizable. To this end, an extension to the Eclipse preferences must be made, linked to the preferences in the GOAL core. As the GOAL preferences are based on a single file, the path to such a file will have to be set. To prevent permission issues, this path is set to the current Eclipse workspace (e.g. the path in which the projects also reside) at the initialization of the plug-in. The registration and initialization of the preference store within Eclipse has been discussed at the end of the previous section, but their coupling to the preferences editor within Eclipse itself has not been discussed yet. This is done through the `org.eclipse.ui.preferencePages` extension point, which lists a main GOAL preference category and three pages within that category: `GoalRuntimePreferencePage`, `GoalLoggingPreferencePage`, and `GoalTemplatePreferencePage`. The runtime and logging preferences pages are both linked to settings from the GOAL core; a key-value pair there is tied to a key-value pair within the Eclipse preference store, and synced automatically through functions in the abstract class. Eclipse also provides a 'restore defaults' functionality, which that abstract class also provides. Both pages make use of classes from the `org.eclipse.jface.preference` package, like `BooleanFieldEditor`, `ComboFieldEditor`, and `FileFieldEditor`. These classes automatically provide a graphical interface of the type linked to the corresponding preference key and a certain description. However, this package has no default functionality to group these fields into (visible) categories. An implementation for this could be found online, which explains the existence of the `GroupFieldEditor` class in our code. The template preference page is generated automatically by DLTK.

An example of the resulting interface is illustrated in Fig. 5.7.

Requirement: *Customization to a user's preferences (6)*

Help Eclipse also provides a native help function, to which a plug-in can make an extension through the `org.eclipse.help.toc` extension point. This extension point requires references to Table of Content (*TOC*) files, which are *XML* files included in the plug-in. These index files, in turn, link to *HTML* pages that compose the actual help content. The GOAL help currently contains some pointers to the available online documentation, as illustrated in Fig. 5.8.

Requirement: *Improved language learning support through (help) documentation (10)*

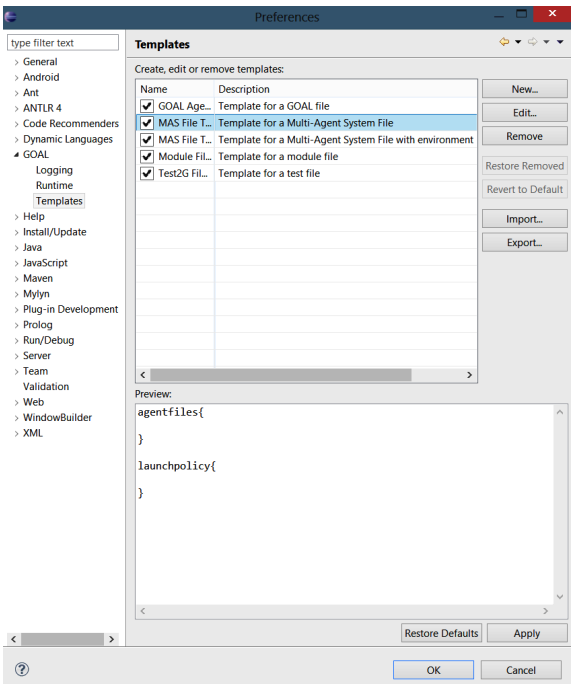

Figure 5.7.: The Templates category of the GOAL preferences

Class overview An overview of the classes that were discussed in this section is given in Tab. 5.3.

With the classes discussed in this chapter, multiple requirements related to the framework for the plug-in and its editor are fulfilled. In the next chapter, the requirements related to running and debugging a system will be handled.

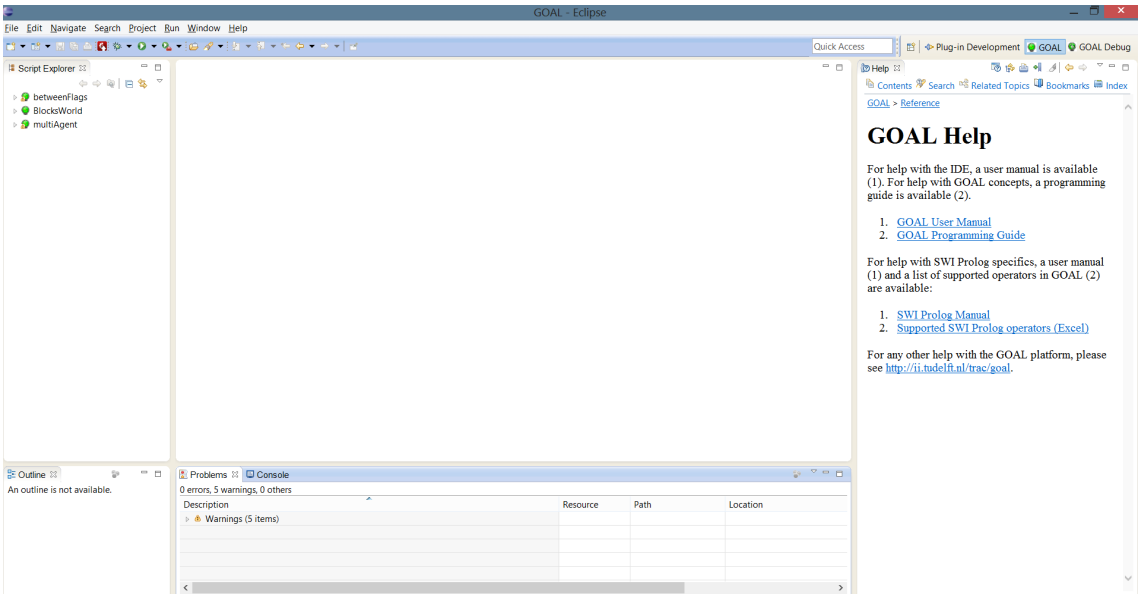

Figure 5.8.: Contents of the GOAL Help page

Class	Package	Extends/Implements
GoalCompletionEngine	completion	org.eclipse.dltk.codeassist. ScriptCompletionEngine
GoalCompletion ProposalComputer	completion	org.eclipse.dltk.ui. text.completion. ScriptCompletion ProposalComputer
GoalCompletion ProposalCollector	completion	org.eclipse.dltk.ui. text.completion. ScriptCompletion ProposalCollector
GoalCompletionProposal	completion	org.eclipse.dltk.ui. text.completion. ScriptTypeCompletionProposal
GoalOverride CompletionProposal	completion	org.eclipse.dltk.ui. text.completion. ScriptTypeCompletionProposal
GoalCompletionProcessor	completion	org.eclipse.dltk.ui. text.completion. ScriptCompletionProcessor
GoalAutoEditStrategy	editor	org.eclipse.jface.text. DefaultIndentLine AutoEditStrategy
GoalContentAssistPreference	editor	org.eclipse.dltk.ui. text.completion. ContentAssistPreference
GoalDocumentationProvider	search	org.eclipse.dltk.ui.documentation. IScriptDocumentationProvider
GoalFoldingBlockProvider	prefs	org.eclipse.dltk. ui.text.folding. IFoldingBlockProvider
GoalFoldingCommentProvider	editor	org.eclipse.dltk. ui.text.folding. PartitioningFoldingBlockProvider
GoalPreferencePage	prefs	org.eclipse.jface.preference. FieldEditorPreferencePage
GoalLoggingPreferencePage	prefs	GoalPreferencePage
GoalRuntimePreferencePage	prefs	GoalPreferencePage
GoalTemplatePreferencePage	prefs	org.eclipse.dltk.ui.templates. ScriptTemplatePreferencePage
GroupFieldEditor	prefs	org.eclipse.jface. preference.FieldEditor

Table 5.3.: An overview of all classes discussed in this section

6. Debugging Environment

Designing a debugger for a rule-based language like GOAL is not the same as designing a debugger for i.e. an object-oriented language like Java, on which most examples that do exist are based. Even just stepping through the code cannot be done in the same way, as the unification process works very differently compared to the more linear code evaluation of other programming languages. Users want to see different, specific information for which custom solutions will have to be built. Moreover, in the previous chapter, problems with the documentation for Eclipse and DLTK were identified, as this documentation is often incomplete or outdated. However, for debugging, this problem is even worse. There is only one document related to implementing a debugger available, but this is an IBM guide from 2004[1] that contains many outdated examples. DLTK contains a substantial debugger framework as well, but there is no documentation for it available at all. There is a big need to provide documentation and guidance for implementing a debugger using the DLTK framework; this documentation will be valuable to others who want to implement a debugger using the DLTK framework.

6.1. Framework

The debugging environment is composed of many sub-systems. Before explaining the actual implementation, an overview of the structure of the implementation will be given in this section.

In order to create an interface for a user to running or debugging a system, a launch configuration is needed. A launch configuration is a set of values that are needed to run a specific system, e.g. containing the path to the file that is to be executed. These configurations are created by such a launch shortcut, and automatically saved by Eclipse, allowing users to re-use such a configuration to launch a system in the same way. These saved configurations are accessible through a drop-down menu next to the run or debug button in Eclipse; the buttons themselves execute the most recently saved launch configuration, or the launch shortcut of the currently selected file or project. These launch shortcuts are also available when right-clicking on a file or project (through the 'Run As' and 'Debug As' menus).

When a user runs a system, an 'execution engine' process is created that provides an interface between Eclipse and yet another process that is actually executing the

[1] https://www.eclipse.org/articles/Article-Debugger/how-to.html

file: the interpreter. An interpreter is a program that executes instructions written in a programming language without previously compiling them into machine code. GOAL fits this description, as it is run through Java, using the source code directly. The separation of concerns between the execution engine and the interpreter does not only allow a great amount of code re-use, it also decouples the user interface, the communication, and the actual execution from each other, allowing all parts to continue working even when i.e. the execution has crashed or is processing for a long time and improving maintainability. Moreover, the debugger engine, providing the interface between the IDE and the interpreter or debugger, depends on DLTK and DBGP, and thus the whole Eclipse framework. An interpreter is part of a programming language itself; GOAL can, for example, be used through a command line interface as well. Making a programming language depend on Eclipse packages is far from ideal; this bloats the command-line version of GOAL for example, and creates a circular dependency structure.

For debugging a system, instead of an interpreter, a debugging engine is required. Such an engine has to provide support for many more functionalities than only executing a file. In other words, it uses an interpreter to execute the code, but has to provide specific debugging functionalities as well. In our case, these functionalities that are required originate from DBGP. The DLTK debugging framework is based on DBGP: a common debugger protocol for languages and debugger UI communication, created in 2003[2]. This protocol is intended to communicate between a debugger engine and an IDE. It is meant to be extendable for language specific features, and supports both dynamic and compiled languages, possibly with multiple processes or threads. As we did not want to 'reinvent the wheel', some other projects that use DLTK were evaluated, as explained in the previous chapter. Amongst these projects is Freemarker[3]: a tool for combining Java and HTML using template code. This project is partly open-source, and this open-source part contains a framework for running and debugging projects using DLTK and DBGP, which is used by our implementation, both for running and debugging a system, as it provides almost the entire execution engine.

In the following sections, the implementation of the mentioned elements will be discussed.

6.2. Running a MAS

The DLTK framework does not know how to execute a multi-agent system, and needs to be able to delegate this task to the GOAL core. In this section, the connection of the support for running a MAS to the DLTK framework will be discussed, and the realization of this functionality for running and terminating a MAS within Eclipse will be explained.

[2]http://xdebug.org/docs-dbgp.php
[3]http://freemarker.org

Launch configuration A launch shortcut has to be defined through the `org.eclipse.debug.ui.launchShortcuts` extension point. This class specifies the possible run modes (run and debug) and ties our plug-in's nature to the `GoalLaunchShortcut` class, which in turn links the nature to the relevant launch configuration type, which is defined through the `org.eclipse.debug.core.launchConfigurationTypes` extension point. This extension point requires specifying a 'source locator' and a 'source path computer'. These classes deal with file and project management, and are provided by DLTK itself. We need to provide specific implementations for our project, and thus the `org.eclipse.debug.core.sourceLocators` and `org.eclipse.debug.core.sourcePathComputers` extension points are used to define these implementations, directly linking to classes from DLTK. Finally, a 'delegate' has to be defined that is responsible for the actual launching: the `GoalLaunchConfigurationDelegate` in our case. This class ensures that the interpreter registered for our plug-in will be used to run the file.

A launch configuration can also be edited once it has been created. To this end, an interface that allows changing relevant settings has to be provided through the `org.eclipse.debug.ui.launchConfigurationTabGroups` extension point, linking our launch configuration type to the `GoalLaunchConfigurationTabGroup` class. This class provides one or more tabs for the configuration editor to use. In our case, these are two tabs. The first tab is the `GoalLaunchConfigurationTab`. This tab contains a selector for the project to use and the actual file to execute within that project. Our specific implementation only has to specify which files are valid for execution, as this can be selected in a dialog. The second tab is DLTK's `ScriptCommonTab`, containing general settings like where the launch configuration is saved, whether a console should be used, whether the output of that console should be logged to a file, etcetera. Together with the existing general preference panes as discussed in the previous chapter, the launch configuration forms the full set of parameters that are user-customizable.

Interpreter To register an interpreter with DLTK, the `org.eclipse.dltk.launching.interpreterInstalls` extension point can be used to define a collection of `InterpreterInstallTypes`, which are in turn defined through the `org.eclipse.dltk.launching.interpreterInstallTypes` extension point. In our case, this links to the `GoalInterpreterInstallType` class. This install type, in turn, links our plug-in nature to the `GoalInterpreterInstall` class, This class is intended to run a file through an executable defined in the install type. However, because GOAL is Java based, there is no specific executable that can do this. Therefore, the `getInterpreterRunner` method is overridden in order to implement our own 'interpreter runner' (execution engine) that does something different from this default behavior: the `GoalInterpreterRunner`.

Execution engine The sole task of the aforementioned `GoalInterpreterRunner` is to create a `GoalRunnableProcess` using the information passed from DLTK: an `IInterpreterInstall`, an `ILaunch`, and an `InterpreterConfig`. The interpreter install is of course our own `GoalInterpreterInstall`, and the interpreter configuration contains parameters specific to the launch the user requested, like the path to the file that is to be launched. The `ILaunch` object is used by Eclipse and DLTK internally as a store for specific objects created in the debugging process. The `GoalRunnableProcess` class is responsible for creating a specific thread for the engine, managing its life-cycle and handling the interface that thread has to an external process (e.g. the input and output). Thus, creating and stopping a process and displaying its output is handled here. This external process has to be created by the actual implementing class, so our `GoalRunnableProcess` is responsible for creating the aforementioned GOAL Java process.

In order to do this, first, the path to the GOAL *jar* file has to be determined. This file has been included as a dependency in the plug-in, as it relies on multiple classes from the GOAL core. Therefore, the path to this file can be found using Eclipse's `FileLocator` class. Using this path and Java's `ProcessBuilder` class, we can launch the *jar*, passing it specific parameters and setting the correct environment variables for SWI Prolog to use. However, some class within the *jar* file needs to be executed. To this end, in the GOAL core, a `goal.tools.eclipse` package has been created that contains several classes responsible for interfacing between the GOAL core and Eclipse. For running a MAS (or a unit test), only the `RunTool` class in this package is needed. This class has only one main method in which a few simple steps are taken:

1. Load the preferences through a file that is indicated by the first passed argument.
2. Parse the *mas2g* or *test2g* file that is indicated by the second argument.
3. If valid, run the MAS or unit tests, forward all output from the GOAL core whilst waiting for it to complete, and terminate the process afterward.

The output of this process is captured and processed by the classes, which allows the user to view it in a native Eclipse console. This console also allows the user to forcefully terminate the process, which can be done using the created Java `Process` object.

Finally, for the sake of user convenience, the `GoalRunnableProcess` class contains some code that determines which file to actually execute. This is because a user can select the run-action on any file within a GOAL project, or even on the project itself, but only *mas2g* and *test2g* files are actually executable. Thus, if the file is not of such an extension, the project that file is in is searched for files of those extensions. If just one matching file is found, it is directly used. Otherwise, a dialog is presented to the user that lists the matching files, allowing the user to select the one to actually execute.

Requirement: *The ability to run a system whilst inspecting or logging its output (5)*

Requirement: *The ability to execute a unit-test whilst inspecting or logging its results (11)*

Embedded language As mentioned before, GOAL contains an embedded KR language, SWI Prolog by default. However, SWI Prolog is platform-dependent, and using this from Java is not trivial. GOAL is originally delivered with custom builds of SWI Prolog 6.0.2 for different platforms: Windows (32 and 64 bits), Mac OS X (64 bits), and Linux (64 bits). All of these platforms require specific SWI Prolog libraries: *DLL* files for Windows, *DYLIB* and *JNILIB* files for Mac, and *SO* files for Linux. In addition, for each platform, there is a different set of files that SWI Prolog uses for its native functionalities, like calculations, reading and writing files, etcetera. The standalone version of GOAL uses its installer to detect a set of specific platforms, and installs only the correct files into the program. Of course, the standalone version of GOAL is a Java program too, and loading these libraries is again not trivial, mostly because the different libraries have dependencies on each other. On Windows, for example, the *BAT* or *EXE* file that starts GOAL adds the folder with the required *DLL*s to the system's `PATH` variable and Java's `java.library.path` variable, which allows Java (and Windows) to find and load the libraries. Moreover, the `SWI_HOME_DIR` environment variable is set, which in turn allows SWI Prolog to find its native libraries. On Linux, a similar *SH* file is used to add the path of the required *SO* files using the `LD_LIBRARY_PATH` environment variable, and the same Java library path is set. Mac is nearly identical to Linux, but the environment variable is named `DYLD_LIBRARY_PATH`. However, for our plug-in, it is impossible to do this. Eclipse is a Java application that uses its own environment variables to begin with, and our (Java) plug-in is loaded into its (Java) environment; we cannot use any script file to set environment variables!

All required SWI Prolog files for all platforms were included in the plug-in, split into different folders for the different platforms. From Java, it is possible to determine the operating system and its so called arch (32 or 64 bits) in a very general way. Eclipse even provides a more accessible interface to these native Java features through the `org.eclipse.core.runtime` package. A new class called `KRtools` was created within the plug-in, and its initialize function is called on start-up of the plug-in. The main goal of the class is to load the Java Prolog Link (JPL) library and initialize it, which allows Java to communicate with the native SWI Prolog library. The JPL library has a dependency on a SWIPL library, which in turn has some more (platform-dependent) dependencies. The initialization of JPL mainly consists of setting the directory for SWI Prolog to use, which as mentioned before can be done through a `SWI_HOME_DIR` environment variable. However, it is also possible to pass this manually by changing the initialization arguments of JPL, and adding a `--home=path` option in there. In order to correctly load all of the required libraries, the initialization function first determines the operating system in a general way, e.g.

detecting if the platform string contains the words "*win*" or "*mac*", and defaulting to Linux otherwise. This is a very flexible way that works for a wide range of platforms without having to explicitly specify each and every one of them like in the original installer.

After the correct platform has been determined, the directory of the specific native libraries to that platform is added to the `java.library.path` property, which is not that difficult using the Java Reflection API. However, loading the right libraries into the system itself is a bit more difficult, and different per platform. On Windows, this is relatively easy: when using the full path to a required *DLL*, and loading the libraries in the right order of dependencies, the system is automatically able to resolve the dependencies between the libraries. Thus, on Windows, only four ordered calls to `System.load` are required in order for everything to work. On Linux and Mac, although loading libraries through their full path is possible, the dependencies are not automatically resolved like this, unfortunately. Multiple solutions for this problem exist.

On Mac, each user has a folder that is automatically searched to resolve dependencies: `user.home/lib`, where `user.home` is a system property indicating the home folder of the current user. As this folder is always writable by the current user, it is sufficient to copy-paste the required libraries into this folder in order to let the operating system find them. On Linux, however, no such folder exists, but it is possible to add a folder to the search path for dependencies through adding a file with that path as its content in the `/etc/ld.so.conf.d` folder. However, this folder is not writable by default, and the system function `ldconfig` has to be called afterward in order to refresh the search path of the system. Both steps thus require administration right, and forces Eclipse to be run as root on Linux.

Aside from the required admin rights on Linux, these solutions are also lacking for another reason, as the SWI Prolog library depends on two very common libraries on Linux and Mac: `readline` and `ncurses`. These dependencies only work with specific versions of the libraries, and thus those are included. However, when the version that another program or the OS itself requires is different from that, problems arise. Especially when using the `ld.so.conf.d` directory in Linux, it is possible to overwrite any original library links, and potentially cause the whole OS to malfunction. Another solution for finding the dependencies of a specific library works on a low level, and involves the Executable and Linkable Format (ELF): the file format used for libraries on Unix-based systems. A library in the ELF format contains a section called 'dynamic', where the dependencies on other libraries are defined. Here, the aforementioned `LIBRARY_PATH` variables play a role, as when a dependency cannot be found through the entries in the 'dynamic' section, the paths in those environment variables are automatically searched to resolve the dependency, which never includes the folder the library itself is in by default. However, it is possible to change this using a tool called `patchelf` (on Linux) or `install_name_tool` (on Mac). Using these tools, it is possible to set the default look-up path of a dependency, the run-time search path (`rpath`), to the folder of the library itself by including the

variable `$ORIGIN` (on Linux) or `@loader_path` (on Mac) in that path. These variables are automatically translated into the library's current path when it is loaded. These settings can also be used when linking the libraries (at compile time), but recompiling the libraries takes a lot more effort than using these tools. This solution does not require copy-pasting any file or set any environment variables, as all dynamic links are now automatically and locally resolved. which in turn prevents any possible conflict with other applications.

Class overview An overview of the classes that were discussed in this section is given in Tab. 6.1. The `org.eclipse.gdt.launching` and `org.eclipse.dltk.dbgp.debugger` packages are both based on the Freemarker framework.

Class	Package	Extends/Implements
GoalLaunchShortcut	launch	org.eclipse.dltk.internal. debug.ui.launcher. AbstractScriptLaunchShortcut
GoalLaunchConfiguration Delegate	launch	org.eclipse.dltk.launching. AbstractScriptLaunch ConfigurationDelegate
GoalLaunchConfiguration TabGroup	launch	org.eclipse.debug.ui. AbstractLaunchConfiguration TabGroup
GoalLaunchConfigurationTab	launch	org.eclipse.dltk.debug. ui.launchConfigurations. MainLaunchConfigurationTab
GoalInterpreterInstallType	launch	org.eclipse.dltk. internal.launching. AbstractInterpreterInstallType
GoalInterpreterInstall	launch	org.eclipse.dltk.launching. AbstractInterpreterInstall
GoalInterpreterRunner	launch	org.eclipse.gdt.launching. AbstractRunnable InterpreterRunner
GoalRunnableProcess	launch	org.eclipse.gdt.launching. DLTKRunnableProcess
FileTool		

Table 6.1.: An overview of all classes discussed in this section

6.3. Integrated debugging

In this section, first, an overview of the DBGP specification will be given, according to the official documentation. Next, as the link between DBGP concepts and GOAL is not always clear, the debugging process for GOAL specifically will be discussed. Finally, the details for the actual implementation will be given.

DBGP Debugger The protocol defines five states a debugger engine can be in:

1. *Starting*: the state prior to the execution of any code.
2. *Stopping*: the state after the code execution has completed, which allows the IDE to further interact with the debugger engine to, for example, collect performance data.
3. *Stopped*: the IDE is detached from the debugger process, no further interaction is possible.
4. *Running*: code is currently executing.
5. *Break*: the code execution is paused and the IDE/debugger can pass information back and forth.

For passing information back and forth in all of these states, a communication scheme has been defined. The IDE sends simple ASCII (plain text) commands to the debugger engine, but the debugger engine is required to respond with XML data. This difference has been made to avoid the debugger engine having to parse XML, for which additional libraries would be required. Moreover, the communication is usually in the form of command (from the IDE) and response (from the debugger), often with larger and more structured information in such a response. Java sockets are used to perform the actual communication, optionally even allowing a debugger engine to run on a different machine from the IDE.

In the starting state, the debugger engine is supposed to send an initialization packet to the IDE, passing its identification and a unique session identifier along. The IDE should then respond with any of the available commands. Usually, the IDE should first respond by sending so called 'feature packets'. This is done in order to determine which features the debugger engine supports, which can range from the used string encoding to the breakpoint types that exist and more. Additionally, a debugger can indicate if it supports asynchronous operation or not. If a debugger does not, the IDE will not be able to send a break command to the debugger engine; only the debugger engine will determine when the execution is running or not. As we want to allow a user to pause the execution from Eclipse, our debugger will support asynchronous operation.

After the feature negotiation, either the running state or the break state can be used as the first state, e.g. automatically running the system or requiring the user to do this. Whilst an asynchronous engine is in the running state, the IDE can send a

break (e.g. pause) command only, requesting the engine to go into the break state. In this state, multiple 'continuation commands' are available:

1. *Run*: starts or resumes the script until a new breakpoint is reached, or the end of the script is reached.
2. *Step into*: steps to the next statement; if there is a function call involved it will break on the first statement in that function.
3. *Step over*: steps to the next statement; if there is a function call on the line from which the command is issued, the debugger engine will stop at the statement after the function call in the same scope as from where the command was issued.
4. *Step out*: steps out of the current scope and breaks on the statement after returning from the current function.
5. *Stop*: ends execution of the script immediately; the engine may be terminated right away, followed by a disconnection of the network connection from the IDE.

Of course, the debugger can change its state on its own as well, e.g. responding to a breakpoint (which will be discussed later on) or the end of an execution.

When the IDE detects the debugger is in a break state, it can request specific information about the code execution state the debugger has halted upon. First of all, the current call stack can be requested. A call stack indicates which subroutine (e.g. part of the code) is about to be executed, and optionally the 'route of function calls' towards that point. This information points directly to the code the user has written, and thus includes file names, line numbers, etcetera. The other information that can be requested is related to the subroutine that is about to be executed: the current context. A context indicates which data will be available to the subroutine at what level: e.g. local variables, global variables, etcetera (this name is up to the debugger engine itself). These variables can have a certain type, other variables as its children (for i.e. an array), and additional parameters like its memory address and size, if it is a constant or not, etcetera. Languages may have different names or meanings for data types, but an IDE may want to be able to handle similar data types as the same type. For this reason, a minimal set of standard data types is defined by DLTK, and a method for specifying more explicit facets on those types. A mapping is required from the engine in order to map the language data types to these common types. The objects from DLTK that should be returned for a certain input string correspond to these common data types and language data types respectively. The explicit facets are additional plain strings that can be passed along with the variable data, like public or private.

A final relevant command that a debugger engine should support is 'eval'. This command allows the IDE to send a string to the debugger engine for evaluation within the current context, like an expression or a code segment to be executed.

The debugger engine should reply to this indicating if the evaluation was successful and an optional response for it.

The entire communication process, including the state management, parsing commands from the IDE, creating XML replies, etcetera, is done by the Freemarker framework. We have to provide the settings for the supported features and an implementation of an `AbstractDebugger`. Such a class will need to have an implementation of the functions `doRun`, `doStop`, `resume`, `suspend`, `collectVariables`, `createBreakpoint`, and `removeBreakpoint`, which correspond with the different commands an IDE can give to the debugger.

Debugging GOAL As mentioned several times before in this section, it is not instantly clear how the features that DLTK and DBGP provide are related to debugging a rule-based language like GOAL. The actual meaning of terms used in the previous part like function call, scope, call stack, and more are not clear in this context. However, based on the composed requirements for our debugger and these features, a mapping can be created. In some instances, this can be done one-to-one, but in others, the IDE will have to be extended with custom implementations.

> ***Requirement**: Show the state of all agents in a system and allow running, pausing, or killing an agent (15)*

As DBGP supports debugging multiple threads in a single program, each agent can be represented by a thread, which it also is within the GOAL environment. In this way, run, pause (break), and kill (stop) actions are automatically provided for each individual thread, and thus each individual agent. A 'debug model' can be provided in order to customize the presentation of these threads, in order to display the agent's state for example.

> ***Requirement:** Allow convenient inspection of an agent's mental state (17)*

When a thread is in a paused state, Eclipse requests the current position in the code (call stack) from the debugger engine, to be displayed within the relevant source file itself. In addition, the current context is requested, e.g. the data that is currently available on different levels. Both of these requests are described by DBGP, but it is unclear how the proposed context levels like local and global correspond to the data that is available for a GOAL agent (goals, beliefs, etcetera). In addition, the applicability of constructs like data type or additional facets is not evident, as terms and atoms are very different concepts than booleans and integers. The current GOAL IDE shows an agents state by using multiple tabs for the different kinds of data: beliefs, goals, mails, and percepts. The presentation of the content in each of the tabs is the same. Other related APL IDEs use a similar presentation. These data level are significantly different from e.g. global or local variables, as not only their scope is different, but their functionality as well. Therefore, the presentation in different tabs seems very suited, and well need to be incorporated into Eclipse.

However, in addition to an agent's mental state, variables can be used as well. The current IDE does not show any of those 'local' variables at any time. However, in order to understand what a certain part of code is doing, information about the instantiation (unification) of those variables is vital. Thus, an additional mechanism similar to the mental state inspection is required.

The default interface of a context provides several features, like allowing a user to search the data. When adapting these mechanisms for the use as discussed above, the existing features will have to be migrated as well whenever possible.

Requirement: *Support stepping through the code of an agent (16)*

Eclipse automatically enables step-actions for a paused thread: step into, over, and out. However, again, the DLTK documentation discusses function calls, whilst such a concept does not exist within GOAL. However, it is possible to translate the basic ideas behind these different step actions into mechanisms specific to GOAL. The step into action, for example, is intended to follow the execution order of the program. The execution process will never be changed by the stepping process; the different step actions only determine which parts the user gets to see (e.g. are halted upon).

In the case of GOAL, we need to carefully determine what parts the user wants to halt upon, as there is no linear code evaluation. It would be possible, for example, to step on each step in the unification process, similar to the trace functionality of Prolog. However, this would require an enormous amount of stepping actions, with little benefit to the user. Instead, the stepping process should follow the way in which a user has written his program, displaying a relevant set of variable instantiations for each step. To this end, a fixed set of code stepping points have been made available, as shown Fig. 6.1. By default, they are not all enabled; this can be changed in the plug-in's preferences. The default setting ensures that the stepping will follow the code order as much as possible, without jumping around a file or halting on 'basic' mechanisms like unification or inserting/deleting a belief. Other options are available to support specific situations. An example of such as situation occurs when using a synchronous environment: an environment that does not change (i.e. continue) without the agent interacting with it. In such a setting, it might be useful to break after an action has been executed, in order to inspect the new state of the environment.

Thus, using these code stepping points, the results of the different step actions can be defined. Logically, step into follows the enabled code stepping points. The step over action is intended to skip the current piece of code and continue to the next in the same scope (module). For GOAL, this can be translated to skipping the action part of a rule (on 'evaluation of rule conditions'), the entry of a module (on 'call to a module'), or a whole module (on 'entry of ...'). The step out action is similar, but should break out of the current scope (module). Thus, when in a module, or about to execute it, the whole (remainder of the) module will be executed, breaking when the previous module on the stack is reached. When there is no other module on the stack (being in the main module for example), the debugger will break when

re-entering that module, as there is no other module to return to. Thus, the step out mechanism can be used to make an agent perform one entire cycle, similar to the stepping mechanism in the old IDE.

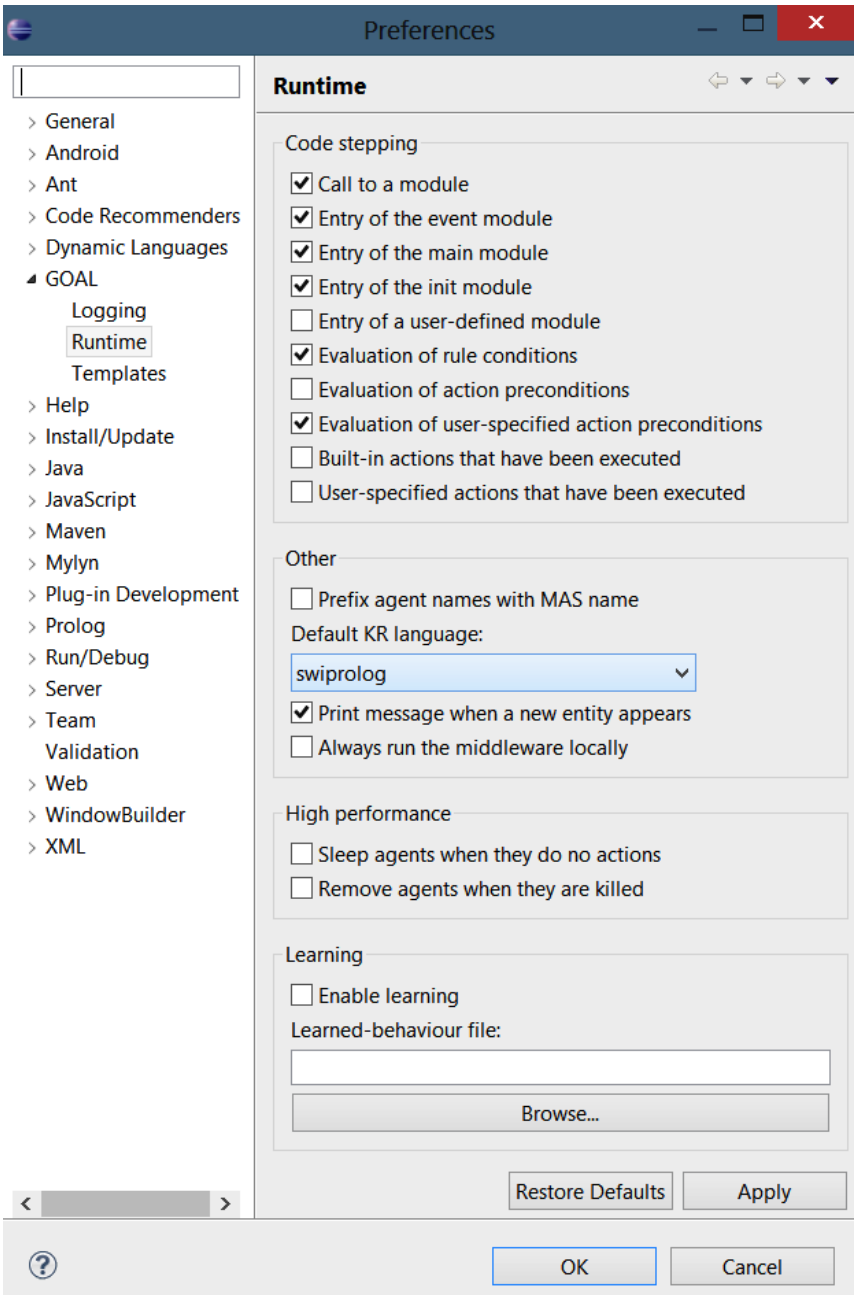

Figure 6.1.: The GOAL runtime preferences, showing the available code stepping points

Requirement: *Support posing queries to an agent (18)*

Eclipse provides an evaluation interface by default, and DBGP contains an evaluation command as well. This interface consists of a console where the user can enter a query, after which the response from the debugger engine is shown. The current GOAL IDE contains a very similar interface, but with different buttons for an informative query or an action command. However, recognizing when the user's input is an action or a query is possible, and thus the default interface can be used without significant alteration.

Requirement*: Support adding or removing breakpoints during or before execution (19)*

Setting breakpoints is a functionality offered by Eclipse by default, but the different types will need to be registered with the plug-in and made available to the user properly. In the current IDE, breakpoints are specific to rules. Setting a breakpoint at a certain line will make the execution halt at the rule at that line or the closest line after it. A conditional breakpoint does the same, but only when the rule actually applies. Although it might be more logical to make the available breakpoints correspond with the available code stepping points, setting breakpoints is only available on a line-based fashion in Eclipse. A large amount of breakpoint types would be required in such an interface to support all code stepping points, and all with their own image for identification by the user. As this is impractical, the current breakpoint system was used. However, more research might have to be done in order to perfect this system.

To finalize the support for breakpoints, the breakpoints created in Eclipse will need to be passed to the GOAL core at its initialization, and a mechanism to update them in the GOAL core when a change is made in the IDE will have to be put in place.

Implementation foundation Through the `org.eclipse.dltk.debug.scriptDebugModel` extension point, a unique identification for our 'debug model' is created. Such a model is used to set and store some preferences related to the debugging process that DLTK supports. The next extension point of the debugging foundation is `org.eclipse.debug.ui.debugModelPresentations`, which links the debug model to the `GoalDebugModelPresentation` class. This class is responsible for generating the text and images shown to a user whilst debugging, based on objects that are used internally by DLTK. Besides this presentation, a 'UI toolkit' has to be defined through the `org.eclipse.dltk.debug.ui.language` extension point. For this extension, we use the `GoalDebugUILanguageToolkit` class, providing default DLTK functionality and linking to our debug model. The final extension point of the debugging foundation is `org.eclipse.dltk.launching.debuggingEngine`. First of all, an identification for our engine is defined by the `GoalDebuggingEngineSelector` class, which has a single function that returns the identifier. Next, a class responsible

for creating the debugging engine is defined, which is `GoalDebuggerRunnerFactory`. This class has a single function that creates an instantiation of the `GoalDebuggerRunner`, which is a similar class to the `GoalInterpreterRunner` as described in the previous section. This class also relies on functionality from the Freemarker debugging framework, and thus, besides linking to debugging engine by its identifier, its only task is to create a `GoalDebuggerRunnableProcess`, which is again similar to the aforementioned `GoalRunnableProcess` class. The Freemarker class deals with the management of that process, whilst the specifics of what that process does are determined within our class. The `GoalDebuggerRunnableProcess` is responsible for creating a `DbgpDebugger`, which is also based on a class from the Freemarker extension. Those two classes together form the actual debugger engine that is required.

Communication with the GOAL core In order to facilitate the actual implementation of the several debugging commands that exist, our `AbstractDebugger` implementation needs to be able to start and stop a GOAL process in a similar fashion to the regular run mode. However, it also needs to be able to directly communicate with this process in a bidirectional way. First of all, as each agent is represented by a separate thread, and each thread is assigned a separate debugger by DLTK and DBGP, two `AbstractDebugger` implementations exist within the plug-in: `LocalDebugger` and `ThreadDebugger`. The `ThreadDebugger` is a holder that registers the agent it is assigned to, and then passes all commands on to the `LocalDebugger`. Thus, only one `LocalDebugger` exists, with a `ThreadDebugger` for each additional agent. This organization is managed in the `DebuggerCollection` class, which registers the different threads, agents, and debuggers that exist, facilitating e.g. the relevant debugger to be obtained for a certain thread, or the corresponding agent for a certain debugger, etcetera.

The `onRun` function of the `LocalDebugger`, after registration in the aforementioned `DebuggerCollection`, behaves in nearly the same way as the previously discussed `GoalRunnableProcess` class; it even uses the same code (through static function calls) to determine or request a file of the correct file type for the actual debugging and for creating the actual `Process` itself. However, after this, a `StreamWriter` and `StreamReader` are created using the processes output and input streams respectively. The `StreamWriter` facilitates sending a (string) message to the GOAL core, and the `StreamReader` responds to (string) messages from the GOAL core. The `DebugTool` class in the GOAL core, similar to the `RunTool` of the run-mode, is the main entry point of our debugging process, and contains a similar mechanism. Upon execution (e.g. calling its main method), it takes these steps::

1. Load the preferences through a file that is indicated by the first passed argument.

2. Parse the *mas2g* file that is indicated by the second argument.

3. Load the breakpoints as indicated by the third argument (using the aforementioned `GoalBreakpointManager` class).
4. If valid, create a runtime for the MAS with debugging enabled, attaching an `EclipseEventObserver` to it (both will be discussed later).
5. Create an `InputReaderWriter` using the default in- and output streams (similar to the `StreamReader` and `StreamWriter` classes in the Eclipse part).
6. Execute the runtime, wait for it to complete, and terminate the process afterward.

All communication classes make use of another class in the `goal.tools.eclipse` package: `DebugCommand`. This class represents a message that is send between the debugger and the IDE, and thus standardizes this communication process. It contains a command type (from a fixed list), the relevant agent, and an optional array of data (varying on the command type). The class also contains code that allows for a safe translation to string and back of itself, in order to be used on the raw stream communication between the IDE and the debugger. Using serialization was considered and tested, but the performance of that process was significantly lower than using plain strings. Moreover, the class is structured in such a way that the whole translation process is encoded in about thirty lines of code. A fixed delimiter is used to separate the different parameters, whilst any occurrence of that delimiter within the data is properly escaped. Moreover, a fixed prefix is used in order to identify a debug command; any other output from either side is interpreted as output to show in the general IDE console, which is required as GOAL or any environment can display certain messages to the user. The command itself is delimited by a newline, as for performance reasons, all communication is buffered until the occurrence of a newline, after which the whole line is sent.

The commands that the `DebugTool` in the GOAL core should be able to process correspond with the different functions in the `LocalDebugger` class, which in turn correspond with the different actions a user can take whilst debugging. These are:

- *Run*: resume a paused agent, or restart a killed agent.
- *Pause*: halt the agent before executing the next code part.
- *Step*: execute the current code part, and halt before executing the next.
- *Eval*: evaluate the passed query or action and return a result.
- *Breaks*: update the set of breakpoints using the passed information (which in turn uses the `GoalBreakpointManager` again).

Thus, the `StreamWriter` class the `LocalDebugger` uses can send these `DebugCommands` to the GOAL debugging process, to be processed by the `InputReaderWriter` there, which in turn executes the relevant action. However, the GOAL core itself generates many more messages for the debugger to process in the `EclipseDebugObserver` class. This class is created for a certain agent when it

is initialized, as observed in the aforementioned `EclipseEventObserver`, of which only a single instance exists for a debugging execution. The `EclipseDebugObserver` listens to breakpoint events (of a certain agent) that are present in the GOAL core itself. These events result in messages that are sent to the debugger:

- *Run mode*: a change to the current state of the agent, e.g. when it got killed or when it hit a breakpoint.
- *Mental state*: the insertion or deletion of beliefs, percepts, mails, and goals.
- *Rule evaluation:* the set of assigned variables (substitutions) for the current rule.
- *Action call*: the set of parameters for the action that is about to be executed, including the evaluation for the action's preconditions when applicable.
- *Module call*: the set of parameters for the module that is about to be entered.
- *Module entry/exit*: the entry or exit of a module, either built-in or user-defined.
- *Action executed*: the successful execution of an action.
- *Log:* depending on the settings in the preferences, a message to be shown in an agent's console.

These messages indicate another important mechanism that is in place. As mentioned before, the user needs to be able to inspect an agent's mental state. However, it is infeasible to send the complete mental state of an agent to the debugger every time it is requested, like in the current GOAL IDE. Therefore, all changes to the mental state are sent to the debugger, which thus has its own database on the state of each agent: the `AgentState` class. Instead of the relevant KR objects, like in the GOAL core, this class only contains collections of strings: the actual representations of the variables in the different bases, e.g. as they are shown to the user. Moreover, an agent's module stack is saved here, as is its current state. Although performance increases significantly in this way, there is an inherent chance of synchronization errors. The messages and their handling needs to be designed carefully in order to prevent the agent state in Eclipse from being inconsistent with the actual mental state of the agent in GOAL. However, when this is done, this mechanism can provide the foundation for saving and displaying an agent's state history, which is outside of the scope of this thesis.

Breakpoints Registering breakpoints of a certain type can be done through the `org.eclipse.debug.core.breakpoints` extension point. For each type, a unique identification and a reference to a marker is needed. These markers have been discussed before when dealing with errors and/or warnings in the code (see sec. 5.3); breakpoints are markers as well. Again, markers are simply indications of a certain code location with a certain meaning. This can be an error on that line, a breakpoint of a certain type, etcetera. The only difference is that for the GOAL breakpoints, custom markers need to be implemented. This is done by the

`GoalLineBreakpointMarker` and `GoalConditionalBreakpointMarker` classes, registered in the `org.eclipse.core.resources.markers` extension point. A custom image is defined for each marker using the `org.eclipse.ui.ide.markerImageProviders` extension point; a red stop sign for a line breakpoint, and a yellow stop sign for a conditional breakpoint. These classes are initialized when the user creates a breakpoint, and thus they are passed the relevant file and line-number, for which a marker with a certain description has to be created. The same goes for deleting breakpoints. In order to also store these breakpoints in a fashion compatible with the goal core, the `GoalBreakpointManager` class has been created in the aforementioned `goal.tools.eclipse` package in the GOAL core. Here, the creation or deletion of a breakpoint of a certain type at a certain line in a certain file has to be registered. This class is responsible for transferring and optionally even updating this information from the Eclipse process to the GOAL debugging process.

In order to allow the user to toggle between the different breakpoint types, or plainly adding or removing them, the `org.eclipse.core.runtime.adapters` extension point can be used. This is a very general extension point, that depending on the passed `adaptableType` and `adapter` is able to generate handlers for certain actions of a user. In our case, `org.eclipse.ui.texteditor.ITextEditor` is used as the `adaptableType` with `org.eclipse.debug.ui.actions.IToggleBreakpointsTarget` as its `adapter`, forwarding the action of toggling breakpoints in a text editor to the `GoalBreakpointAdapterFactory` class. This class creates a `GoalBreakpointAdapter` for each event, which in turn creates or deletes the breakpoint for the given file and line number.

By default, changing the type of a breakpoint is only possible through the debugging interface. Additionally, creating a breakpoint takes several actions of the user. However, in order to provide the user with a more intuitive way to do this in the `GoalEditor` class (representing the actual code editor), a listener is added to the `ITextEditorActionConstants.RULER_DOUBLE_CLICK` action: double-clicking in the vertical bar on the left of the code that shows all markers (the `VerticalRuler`). Upon this event, three possible results are possible:

1. When no breakpoint is present at the current line yet, a regular breakpoint is created.
2. When a regular breakpoint is present at the current line, it is deleted, and a conditional breakpoint is created.
3. When a conditional breakpoint is present at the current line, it is deleted (resulting in no breakpoint at all at that line).

This is possible by using Eclipse's `BreakpointManager`, which automatically uses the correct classes as discussed above.

As by default, custom markers are not prioritized, markers for error messages are displayed on top of markers for breakpoints. This is not desired, as breakpoints are

only indicated through such a marker, whilst errors and warnings are also displayed by underlined code. Moreover, breakpoints have a large impact on the debugging process, whilst a warning might not be relevant at all. In order to do this, the annotation type that corresponds to the marker type needs to be customized, e.g. the presentation of the marker needs to be changed from the default. This can be done through the `org.eclipse.ui.editors.annotationTypes` extension point, which can specify a custom annotation type for a certain marker type. These custom annotation types can be defined in the `org.eclipse.ui.editors.markerAnnotationSpecification` extension point. Such a custom annotation contains preferences like if it should be included in the list of all markers, if it should be shown in the vertical ruler, if it should underline the corresponding piece of code, and most importantly for us: on what 'presentation layer' it should be shown. Setting this to a value of '1' will ensure that his marker is always shown at the highest priority. In this way, all required breakpoint functionality is implemented in the most convenient way for the user.

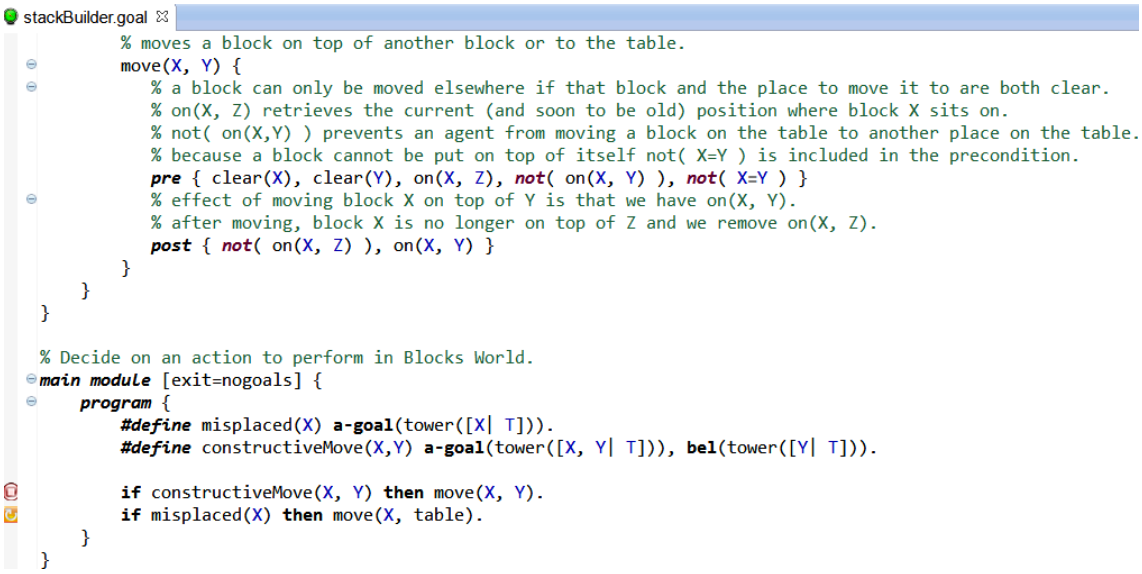

Figure 6.2.: An editor showing a regular (red) and conditional (yellow) breakpoint

Debugger interface The GOAL plug-in defines a custom debug perspective, which is opened when debugging a MAS automatically through our launch configuration. The GOAL debug perspective is registered in the `org.eclipse.ui.perspectives` extension point and organized in the accompanying `GoalDebugPerspectiveFactory` class. In this class, the different user interface elements are created and positioned. Some of these elements are provided by Eclipse or DLTK; others are custom implementations based on Eclipse or DLTK elements. The current organization of this perspective is based on the Java debug perspective and the current GOAL IDE's debug view.

At the top, an overview of the currently running agents is provided. This overview is provided by Eclipse's debug view, and customized by the aforementioned `GoalDebugModel` class in order to show the agent's state, based on the corresponding `AgentState`. In this overview, there is also a tab available where all currently set breakpoints are listed, optionally allowing a user to delete them. Again, this breakpoint view is provided by Eclipse itself. Another 'automatic' view is the one below for the source code. This script view is provided automatically by DLTK, as it is the same as the 'regular' code editor. The currently evaluated line of code is shown by a green marker; the right file is automatically opened at the right location to match this position. This is dependent on the currently selected agent in the overview at the top, as all other views around it are as well. The final view that is provided without customization is the interactive console at the right bottom. Though this view does not change depending on the selecting agent, allowing a user to view the history of posed queries or executed actions, the entered query or command is forwarded to the currently selected agent only. Thus, that agent is also mentioned in the response.

Different, custom consoles exist at the bottom of the screen. By default, Eclipse only has one general console that provides the output for a running program. The GOAL run mode makes use of this console as well, and the debug mode does as well. However, as determined by the set logging preferences, each individual agent can log messages as well. Possibly having many agents, it is infeasible to log these messages to one general console. Therefore, a separate console for each agent is created. To this end, a custom view is registered through the `org.eclipse.ui.views` extension point: the `GoalAgentConsoleView`. For each agent, such a view is constructed when launching the debugging engine for a multi-agent system. This is different to all other views, as this happens during execution, whilst all other views are fixed. This requires these views need to be removed at the end as well. The `GoalAgentConsole` class is used to provide a convenient interface to a console for the debugger; each agent has an corresponding `GoalAgentConsole` registered for it, similar to each agent having an `AgentState`. Next to these agent-specific consoles, a general action history console is also created. In this console, the actions executed by all agents are logged, allowing one to get a quick overview of what all agents are doing within the environment. It is possible to disable all of these consoles in the plug-in's preferences.

As mentioned before, a separate area of the screen has been reserved for inspection of the mental state of an agent. Thus, different tabs for each category have been created. The evaluation of the current line that is shown below is nearly identical to these tabs. Therefore, the abstract `GoalVariablesView` class has been created, with specific implementations for each of the tabs, which are registered in the `org.eclipse.ui.views` extension point as well. This abstract class copies many functionalities from the source of the default Eclipse console class, but has been extensively customized. This was mainly necessary to remove features like the display of a variable's type or scope, but also to allow for the right data to be shown in the tab. As all variables are usually shown in one view, Eclipse and DLTK cre-

ate a single 'stack frame' when execution is halted. However, as the variable views are based on such a stack frame, this frame will have to be customized in order to filter the data for the view. To this end, the `GoalScriptStackFrame` class has been created, taking a regular `ScriptStackFrame` and a `GoalVariableType` as its input. All variables in the frame have been given a type from that class, which allows the right variables to be taken from the frame for the view. Thus, for example, the `GoalVariablesViewBeliefs` takes the `ScriptStackFrame` provided by Eclipse, and translates it into a `GoalScriptStackFrame` with the `BELIEFS` type to be used for that view. When the variables for the view are requested, only the ones of the `BELIEFS` type will be returned, resulting in the correct view for each tab.

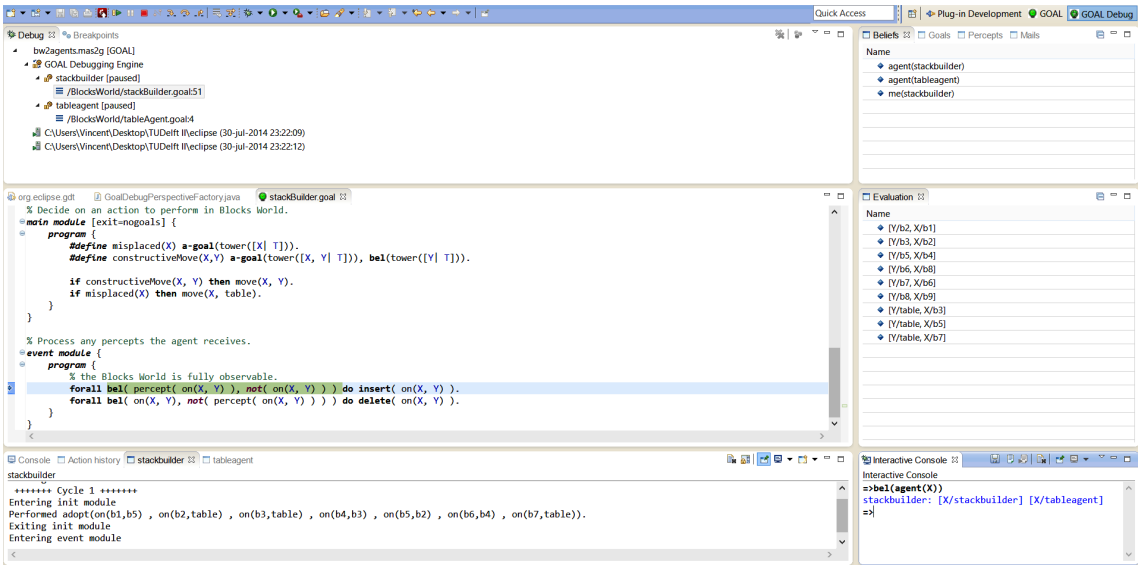

Figure 6.3.: The GOAL debug perspective

Class overview An overview of the classes that were discussed in this section is given in Tab. 6.2.

Class	Package	Extends/Implements
GoalDebugModelPresentation	debug.ui	org.eclipse.dltk.debug.ui. ScriptDebugModelPresentation
GoalDebugUILanguageToolkit	debug.ui	org.eclipse.dltk.debug.ui. AbstractDebugUILanguageToolkit
GoalDebuggingEngineSelector	launch	org.eclipse.dltk.core. DLTKIdContributionSelector
GoalDebuggerRunnerFactory	debug	org.eclipse.dltk.launching. IInterpreterRunnerFactory
GoalDebuggerRunner	debug	org.eclipse.gdt.launching. RunnableDebuggingEngineRunner
GoalDebuggerRunnableProcess	debug	org.eclipse.gdt.launching. DLTKRunnableDebuggingProcess
GoalRunnableProcess	launch	org.eclipse.gdt.launching. DLTKRunnableProcess
AgentState	debug.dbgp	
DbgpDebugger	debug.dbgp	org.eclipse.dltk.dbgp.debugger. AbstractDbgpDebuggerEngine
LocalDebugger	debug.dbgp	org.eclipse.dltk.dbgp.debugger. debugger.AbstractDebugger
ThreadDebugger	debug.dbgp	org.eclipse.dltk.dbgp.debugger. debugger.AbstractDebugger
DebuggerCollection	debug.dbgp	
StreamReader	debug.dbgp	Thread
StreamWriter	debug.dbgp	
GoalLineBreakpoint	debug	org.eclipse.debug.core. model.LineBreakpoint
GoalConditionalBreakpoint	debug	GoalLineBreakpoint
GoalBreakpointAdapterFactory	debug	org.eclipse.core.runtime. IAdapterFactory
GoalBreakpointAdapter	debug	org.eclipse.debug.ui.actions. IToggleBreakpointsTarget
GoalDebugPerspectiveFactory	debug.ui	org.eclipse.ui.IPerspectiveFactory
GoalAgentConsoleView	debug.ui	org.eclipse.ui.internal. console.ConsoleView
GoalScriptStackFrame	debug.ui	org.eclipse.dltk.internal. debug.core.model. ScriptStackFrame
GoalVariableType	debug.ui	Enum
GoalVariablesView	debug.ui	org.eclipse.debug.ui. AbstractDebugView
GoalVariablesViewBeliefs	debug.ui	GoalVariablesView
GoalVariablesViewGoals	debug.ui	GoalVariablesView
GoalVariablesViewMails	debug.ui	GoalVariablesView
GoalVariablesViewPercepts	debug.ui	GoalVariablesView
GoalVariablesViewEvaluation	debug.ui	GoalVariablesView

Table 6.2.: An overview of all classes discussed in this section

7. User Evaluation

In order to monitor the perceived improvement of the GOAL platform by our efforts, several evaluations amongst its users were performed. In this chapter, the used evaluation method will be discussed first. Next, the actual results will be shown, and their implications will be discussed afterward, The full statistical results can be found in Appendix B.

7.1. Method

The System Usability Scale (SUS) is a widely used scale developed in 1996 to quickly and easily assess the usability of a given product or service. This scale has several attributes that make it a good choice for a usability evaluation[35]: the survey is flexible enough to assess a wide range of interface technologies, relatively quick and easy to use by both study participants and administrators, provides a single score on a scale that is easily understood, and free.

A SUS survey is composed of ten fixed statements that are scored on a five-point scale of strength of agreement. Final scores for the SUS can range from 0 to 100, where a higher score indicates better usability. The statements alternate between the positive and negative. The answers to the individual questions are not meaningful on their own; only the emergent score is relevant. This has been confirmed by extensive empirical research[36]. Due to this fact, SUS surveys are often used to compare the usability of different, possibly even dissimilar systems. Products which are at least passable should score above 70, with better products scoring in the high 70s to upper 80s. Truly superior products score better than 90, whilst products with scores of less than 70 should be considered candidates for increased scrutiny and continued improvement. The range of scores is essentially half of the nominal value.

For this thesis, four distinct iterative evaluations have been done. First, the current GOAL IDE was evaluated using first year computer science students from the TU Delft and Leiden University. Next, an evaluation on only the first year computer science students from the TU Delft was done on the first version of the GOAL plug-in for Eclipse. This version contained the complete editing framework, but no custom debugger. Instead, it launched the 'old' debugger interface. After the new debugging framework was implemented, a final evaluation was done on the same group of students. Finally, shortly after the first-year students were finished,

a small group of PhD students used and evaluated the plug-in, resulting in the final evaluation results. All surveys could be anonymously filled in online, with the option to add an additional comment.

7.2. Results

The reliability of the test scores was measured using the Cronbach's Alpha statistic. Being 0.84, the consistency of our survey is suited for low-stakes testing[37].

The first survey on the 'old' IDE had 53 respondents. The 95% confidence interval for the mean score is 43 to 52. The student's comments were about missing auto-completion, differences with modern IDE's, and poor debugging.

The second survey on the first version of the Eclipse plug-in had 64 respondents. The 95% confidence interval for the mean score is 49 to 58. The comments were about the output of unit-tests, missing documentation, and poor debugging again. Unfortunately, some students also used the survey to express discontent with parts of the course they were taking in which they used the IDE, resulting in comments about resits for example.

The third survey on the second version of the Eclipse plug-in had 56 respondents. The 95% confidence interval for the mean score is 39 to 47. The students commented on performance issues, the lack of mental state inspection during execution, and some inconveniences resulting from the use of external module files.

The fourth and final survey on the second version of the Eclipse plug-in had only 6 respondents. Though this is a low number, a trend might be distilled from these evaluations. The 95% confidence interval for the mean score is 64 to 83. The comments were approving of the immediate reporting of errors and warnings and the stepping debugger.

Fig. 7.1 shows the results of all surveys in a single chart. Each survey was roughly five weeks apart, with the exception of the final survey, being taken shortly after the third survey, on a different group of users.

7.3. Discussion

As expected, the initial usability scores for the 'old' IDE, mainly provided by students from Leiden, are not very high. The first version of the plug-in, mainly evaluated by students from Delft, shows a small improvement of this score. However, as the group of participants changed, this small difference is only a slight indication of a positive trend. The second version of the plug-in, including the new debugger framework, dropped slightly below the mean of the first evaluation, even having negative outliers.

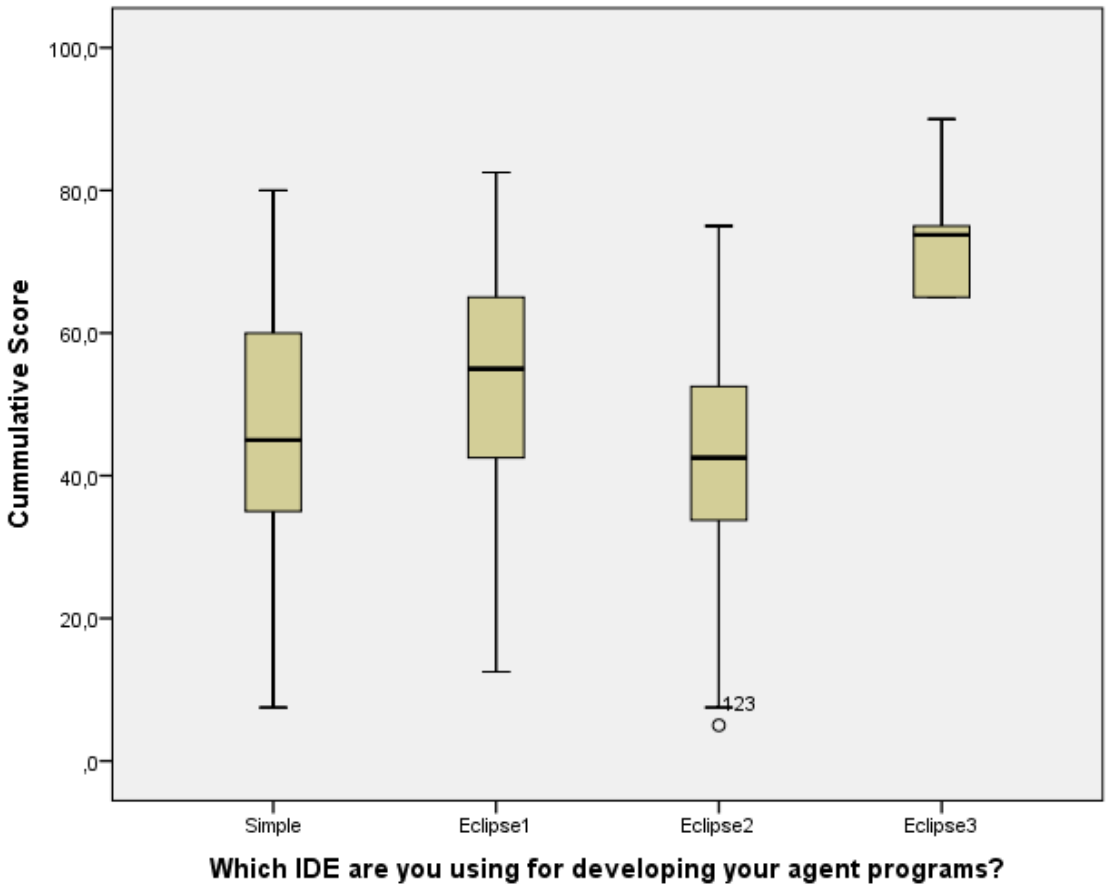

Figure 7.1.: The results from the four SUS surveys

Of course, these results were not foreseen. After all, the plug-in was developed to enhance the user experience. A few possible reasons exist for this drop of the scores:

1. Out of the 150 students taking the multi-agent programming courses, on average just 60 students responded to the surveys. As this is less than half of the user base, the possibility of a significant non-response bias exists. This bias is potentially negative, as the comments have for example shown that students tend to use the survey to comment on unrelated matters they are discontent with, such as exams or lab partners. The negative outliers support this theory, as these correspond with the less IDE-oriented reactions.

2. The addition of the debugger environment was done during the courses. Research suggests that resistance to change can occur when no clear incentive is provided for giving up practices one is accustomed to[38]. Evidence has to be provided to prove the benefits of the new version to prevent this inertia. As the stepping mechanism and the overall visualization changed dramatically, the students needed such evidence, but this was not clearly provided to everyone. The comments on the survey support this theory, as many commented

about the inability to inspect a mental state whilst an agent was running (e.g. not paused). Indeed, this was possible in the old IDE, although those views could be very full, could not be searched or filtered, and refresh faster than one can see. However, people got used to this way of debugging, and thus responded negatively to the removal of this feature. Discussing the new solutions with a user often helped to let them understand the improvements that were made.

3. The IDE is regarded as 'the whole thing'. For example, comments were made about the functionalities of the testing framework. Strictly speaking, this is not an IDE issue. Similarly, comments were made about language constructs. Being the interface to GOAL, the students view the IDE as the whole platform. However, there is still a clear separation between this interface and the GOAL internals. Moreover, because multi-agent systems can make use of external environments, even more factors can influence the user's experience. When there are performance issues, for example, these can be caused by the IDE, the GOAL core, or the environment. However, users quickly regards the IDE as the culprit, being the only accessible part of the software for them, whilst the cause could be in the environment, much further down the pipeline.
4. Some bugs in the new debugger environment were found shortly after its release. Although this is to be expected in a new product, and these bugs were solved quickly, they have influenced the user experience negatively. More extensive testing on different machines could prevent this in the future. Moreover, frequent updates are inconvenient to users, and might even be missed.

The final evaluation, taken on a different set of students, shows a very positive trend. Although based on just a few users, the scores are significantly higher, stimulating further evaluation of the usability of the IDE.

8. Conclusions and Future Work

This thesis discussed the design and implementation of a mature and professional IDE for GOAL. In this chapter, an overview of the contributions of this thesis will be provided. Next, recommendations for future work will be discussed. Finally, a reflection of the results and accompanying conclusions will be given.

8.1. Contributions

A fully functional development environment was developed for the GOAL agent programming language. Through the evaluation of literature and other work in the field, a set of requirements for an agent programming IDE was developed. Based on these requirements and additional evaluation, Eclipse and DLTK were chosen as the framework for our work.

New ANTLRv4 grammars were developed for all relevant file types. This work created a separation of concerns that was not present in the previous grammars, as the code for processing a syntax tree was moved outside of the grammar by using a visitor mechanism, increasing the maintainability and understandability of both the grammar and the processing code. Moreover, by using the new lexical modes, the KR-sections (language islands) are now recognized by the lexer itself, instead of the parser determining what lexer to use. This allows a lexer to be used individually, as required in a modern IDE for performance reasons, and ensures a linear parsing process. In other words, all steps in the pipeline of lexing, parsing, and visiting can be executed independently from each other, only based on the result of the previous step, allowing for improved testability and maintainability. The send construct in the GOAL language was modified in order to facilitate this new process, and the error reporting mechanisms were all unified in a single system.

Next, an editing framework for GOAL was created, providing as many industry standard features as possible, adapted to the context of logic programming where required. Moreover, a debugging framework was created. To this end, the mapping of well-defined debugging processes in other programming paradigms to that of agent programming was evaluated. A key example of this mapping can be found in the stepping process that has been defined, as this is significantly different to the usual linear stepping process for other programming languages, and unique to agent programming. Other AOP-specific features such as mental state inspection and several consoles were designed and implemented as well. Support for breakpoints that

can be updated during debugging was added, together with a fixed communication structure between the different processes. Finally, a reliable method for loading platform dependent libraries from within Java was created.

All of these chapters were also created as an implementation guide for other AOP IDE developers. This works aims to increase the standard of the whole field by providing a complete and adaptable example together with the technical details. Throughout this process, the user experience was evaluated using SUS surveys.

8.2. Future Work

The formulated could-have requirements already indicate opportunities to improve specific tools that are currently offered to a user in the IDE. The support for refactoring, for instance, could be enhanced to include automatic updating of a variable's occurrences when its name is edited, automatic updating of file references, etcetera. In addition, the auto-completion could be made context sensitive: to take into account which suggestions are actually possible in the current context. Defining a section within another section is not possible for instance, but even only suggesting existing beliefs in a `bel(...)` statement should be possible (e.g. not showing goals in the auto-completion there). The automatic formatting could be improved as well, possibly even allowing a user to customize this, as is possible with Java code for example.

Besides improved editing features, there are other areas in which work is required as well. For example, there is currently no documentation format like JavaDoc available for GOAL. Designing and implementing such a documentation structure would not only improve the auto-completion, but also allow for code documentation to be generated, again similar to JavaDoc. Users often create manual documents for this end now, supporting the need for such a framework. A documentation standard might be set for the whole AOP field. Moreover, the testing framework could be improved as well. Although this framework was only finished even during the creation of this thesis, and also outside of its scope, users often commented on this framework in the IDE evaluation. Users regard the IDE as the full platform, and thus every part of it should be as user friendly as possible. The unit test output (e.g. results), for example, is not clear to all users.

More work is also needed in the debugging environment. Another comment that was often given in the evaluation highlighted the inability to properly inspect an agent's state whilst it is running. Pausing an agent is not always ideal, especially in combination with asynchronous environments that keep on going whilst an agent is paused. Although the constantly refreshing panes of the old IDE are not a solution to this problem, more work on this issue is required. A system of watch expressions could be put in place, for instance. This would allow a user to enter specific queries of which the result would be updated continuously. This is not a

trivial task, however, and might not even solve the problem completely, indicating the need for more research on this topic. In addition, the current breakpoint mechanism should be reevaluated as well. The two types of breakpoints that currently exist do not match the new stepping mechanism that is in place, which could confuse a user. Moreover, some potentially useful breakpoints, like after an action has been successfully executed, cannot be set in the current situation. It may be possible to create a mechanism that would allow someone to set breakpoints on specific stepping points. However, more careful evaluation of a user's requirements for breakpoints will be necessary. In addition, mechanisms could be added that allow a user to understand the reasoning of their agents even better. For instance, a navigable history of an agent's states could be created, allowing a user to inspect the conditions under which a certain decision was made, tracing the steps taken to reach that decision at that specific moment. More insight in the unification process of a rule might be provided as well, especially when a rule has failed. It may be possible to show a user exactly why a rule does not apply, or allow a user to request this somehow, instead of only displaying the fact that it does not, like is currently done[39]. However, again, this would require more careful research, possibly even specific to the KR language that is used. This work could also benefit other agent programming languages. Finally, the interaction amongst agents, or even between an agent and an environment, might require more visual support to be made clear to a user. Other APL IDEs already have some mechanisms for this in place, on which a future implementation for GOAL could be based.

A final area of interest for the IDE itself is the dependency system that is currently in place. A module file can be used by multiple agent files, but the semantic validation of such a file depends on its parent. In other words, using an external module file in combination with one agent could cause errors, whilst a combination with another agent would work fine. However, as these errors are shown at their exact position in the code, in the module file in this case, it can be unclear to a user where that error originated from. Improvements to this error system and perhaps a redesign of the dependency model of GOAL agents would be required to solve this problem.

Besides the IDE itself, there are more areas that need improvement. For instance, the evaluation clearly showed the effects of the resistance to change. A more careful release procedure could be determined, perhaps even specific to different user groups. Moreover, as bugs were present in a few released versions of the plug-ins, a more careful test procedure is required. However, Eclipse itself does not provide a framework to test a plug-in conveniently. Nearly all classes in the plug-in depend on the work of others (DLTK, Freemarker, JFace, DBGP, etc.), and thus their specific, local (unit) functionality does not directly correspond to the functionality offered to a user. Some bugs were even specific to certain Eclipse or Java version, which indicates the need for testing in multiple environments as well. A well designed testing procedure, and perhaps even a framework for testing Eclipse plug-ins themselves could help in preventing these bugs, both for our work and that of others. Moreover, the user experience should be continuously monitored as well, again for

different user groups. Although this thesis made a start on these evaluations, they should be performed on larger and less specific user base. In this way, the effect of changes to the IDE can be monitored carefully, allowing for factual improvement in the user experience.

8.3. Conclusions

In an iterative fashion, an IDE for GOAL was created that successfully provides full-fledged editing and debugging capabilities to an agent programmer. The required tooling support for agent program development was determined, and an adequate design of the development environment was created. All must-have and should-have requirements listed in sec. 1.2 have been implemented, although as discussed in the previous section, there is room for improvement on some aspects. All could-have requirements, of which some have been discussed in the previous section as well, are still open for implementation.

Based on the fulfilled requirements, all research questions have been answered in this thesis as well. The specific sub-questions were:

- How can we integrate KR technology in an IDE in such a way that one embedded language can easily be exchanged with another?
- What kind of support is needed in an IDE when working with an external environment during execution?
- Which debugging mechanisms are needed when debugging a rule-based language?
- What kind of debugging support should be provided for debugging agent programs that execute reasoning cycles?

First of all, care has been taken to embed KR technology in the platform in an exchangeable manner. Examples of this are the use of ANTLRv4 language islands and the external ProDT plug-in. Furthermore, to allow a developer to debug an agent program using an external environment conveniently, the execution and debugging environment has been fully integrated within the IDE. Moreover, using DBGP's asynchronous operation mode, all different processes have been carefully separated from each other, ensuring the continuation of one process when another fails. For this debugging environment, the debugging mechanisms applicable to agent programming have been identified and implemented. In the previous sections, suggestions have been given for the development of more advanced mechanisms that may improve the debugging process for an agent program developer even more. Finally, a code stepping mechanism for GOAL, being a rule-based language, has been successfully implemented by using predefined and customizable code-stepping points, and custom implementations for the available step actions in Eclipse.

Care was taken to meet the non-functional requirements in all these steps. For example, the performance of the runtime and debugging environment was constantly evaluated, mostly in comparison to the current situation (in the 'old' IDE). This lead to several refraction processes. For example, the first version of the communication between the GOAL core and the debugger that is composed out of `DebugMessage` objects used the Java serialization framework. However, this object serialization had a considerable performance impact, leading to a redesign to use strings for the communication instead. Lessons learned from the NetBeans prototype also ensured proper performance of the editing framework. Moreover, besides extensive functional testing by myself, the robustness of the implementation was ensured by the direct use of the plug-in by over 150 students and teaching assistants, who helped to identify issues and give suggestions for new features.

The usability of the plug-in was continuously monitored through multiple SUS evaluations. Although the evaluation results were not unanimously positive, possible reasons for this fact were given, and a positive trend was detected. Moreover, even though the adaptation of this work in the field cannot be foreseen, it certainly set a new standard for GOAL by providing it with a mature and professional IDE, currently in use at the TU Delft and several other universities like the TU Denmark and Leiden University by both students and researchers. Several suggestions to improve the user experience and thus the potential adaptation as well are given.

Finally, this thesis contributes to the maintainability of the implementation by providing an extensive guide on its implementation. As extensive care has been taken to allow for other AOP platforms to benefit from this work, a high level of maintainability and extensibility were key factors in the design of the plug-in. For example, the editing framework and debugging environment have been separated in both this thesis and in actual code packages. This separation hopefully allows others in the field to 'cherry pick' the improvements that are relevant to them.

Bibliography

[1] Shoham, Yoav. "Agent-oriented programming." Artificial intelligence 60.1 (1993): 51-92.

[2] Russell, Stuart, and Peter Norvig. "Artificial Intelligence: A Modern Approach." (2009).

[3] Hindriks, Koen V. "Programming rational agents in goal." Multi-Agent Programming: Springer US, 2009. 119-157.

[4] Rex Bryan Kline, Ahmed Seffah, Evaluation of integrated software development environments: Challenges and results from three empirical studies, International Journal of Human-Computer Studies, Volume 63, Issue 6, December 2005, 607-627.

[5] Wasserman, Anthony I. "Tool integration in software engineering environments." Software Engineering Environments. Springer Berlin Heidelberg, 1990.

[6] Ducassé, Mireille, and Jacques Noyé. "Logic programming environments: Dynamic program analysis and debugging." The Journal of Logic Programming 19 (1994): 351-384.

[7] Ossher, Harold, William Harrison, and Peri Tarr. "Software engineering tools and environments: a roadmap." Proceedings of the Conference on the Future of Software Engineering. ACM, 2000.

[8] Waters, Kelly. "Prioritization using MoSCoW." Agile Planning (2009).

[9] Wielemaker, Jan, et al. "Swi-prolog." Theory and Practice of Logic Programming 12.1-2 (2012): 67-96.

[10] Behrens, Tristan, et al. "An interface for agent-environment interaction." Programming multi-agent systems. Springer Berlin Heidelberg, 2012. 139-158.

[11] Joseph Dumas and Paige Parsons. 1995. Discovering the way programmers think about new programming environments. Commun. ACM 38, 6 (June 1995), 45-56.

[12] Tajalli, Hossein, and Nenad Medvidovic. "A reference architecture for integrated development and run-time environments." Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on. IEEE, 2012.

[13] Eric Allen, Robert Cartwright, and Brian Stoler. 2002. DrJava: a lightweight pedagogic environment for Java. In Proceedings of the 33rd SIGCSE technical

symposium on Computer science education (SIGCSE '02). ACM, New York, NY, USA, 137-141.

[14] Kris Powers, Paul Gross, Steve Cooper, Myles McNally, Kenneth J. Goldman, Viera Proulx, and Martin Carlisle. 2006. Tools for teaching introductory programming: what works?. In Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE '06). ACM, New York, NY, USA, 560-561.

[15] Storey, Margaret-Anne, et al. "Improving the usability of Eclipse for novice programmers." Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange. ACM, 2003.

[16] Murphy, G.C.; Kersten, M.; Findlater, L., "How are Java software developers using the Eclipse IDE?," Software, IEEE , vol.23, no.4, pp.76,83, July-Aug. 2006.

[17] Laura Chiticariu, Vivian Chu, Sajib Dasgupta, Thilo W. Goetz, Howard Ho, Rajasekar Krishnamurthy, Alexander Lang, Yunyao Li, Bin Liu, Sriram Raghavan, Frederick R. Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. 2011. The SystemT IDE: an integrated development environment for information extraction rules. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11). ACM, New York, NY, USA, 1291-1294.

[18] Febbraro, Onofrio, Kristian Reale, and Francesco Ricca. "ASPIDE: Integrated development environment for answer set programming." Logic Programming and Nonmonotonic Reasoning. Springer Berlin Heidelberg, 2011. 317-330.

[19] Findler, Robert Bruce, et al. "DrScheme: A programming environment for Scheme." Journal of functional programming 12.2 (2002): 159-182.

[20] Zacharias, Valentin. "Development and verification of rule based systems—a survey of developers." Rule Representation, Interchange and Reasoning on the Web. Springer Berlin Heidelberg, 2008. 6-16.

[21] Zacharias, Valentin. "Tackling the debugging challenge of rule based systems." Enterprise Information Systems. Springer Berlin Heidelberg, 2009. 144-154.

[22] Dastani, Mehdi. "2APL: a practical agent programming language." Autonomous agents and multi-agent systems 16.3 (2008): 214-248.

[23] Collier, Rem William. Agent factory: A framework for the engineering of agent-oriented applications. Diss. University College Dublin, 2002.

[24] Bordini, Rafael H., Jomi Fred Hübner, and Michael Wooldridge. Programming multi-agent systems in AgentSpeak using Jason. Vol. 8. Wiley-Interscience, 2007.

[25] Howden, Nick, et al. "JACK intelligent agents-summary of an agent infrastructure." 5th International conference on autonomous agents. 2001.

[26] Pokahr, Alexander, Lars Braubach, and Winfried Lamersdorf. "Jadex: A BDI reasoning engine." Multi-agent programming. Springer US, 2005. 149-174.

[27] Hirsch, Benjamin, Thomas Konnerth, and Axel Heßler. "Merging agents and services—the JIAC agent platform." Multi-Agent Programming:. Springer US, 2009. 159-185.

[28] Friedman-Hill, Ernest. "Jess, the rule engine for the java platform." (2008).

[29] Bordini, Rafael H., et al. "A survey of programming languages and platforms for multi-agent systems." INFORMATICA-LJUBLJANA- 30.1 (2006): 33.

[30] Bendisposto, Jens, et al. "A semantics-aware editing environment for Prolog in Eclipse." arXiv preprint arXiv:0903.2252 (2009).

[31] Gomanyuk, S. V. "An approach to creating development environments for a wide class of programming languages." Programming and Computer Software 34.4 (2008): 225-236.

[32] Luck, Michael, Peter McBurney, and Jorge Gonzalez-Palacios. "Agent-based computing and programming of agent systems." Programming Multi-Agent Systems. Springer Berlin Heidelberg, 2006. 23-37.

[33] J. Tonn and S. Kaiser. ASGARD – A Graphical Monitoring Tool for Distributed Agent Infrastructures. In Advances in Practical Applications of Agents and Multiagent Systems: 8th International Conference on Practical Applications of Agents and Multiagent Systems (AISC Vol. 70, Springer 2010), April 2010.

[34] Parr, Terence. The definitive ANTLR 4 reference. Pragmatic Bookshelf, 2013.

[35] Brooke, John. "SUS-A quick and dirty usability scale." Usability evaluation in industry 189 (1996): 194.

[36] Bangor, Aaron, Philip T. Kortum, and James T. Miller. "An empirical evaluation of the system usability scale." Intl. Journal of Human–Computer Interaction 24.6 (2008): 574-594.

[37] Santos, J. Reynaldo A. "Cronbach's alpha: A tool for assessing the reliability of scales." Journal of extension 37.2 (1999): 1-5.

[38] Baddoo, Nathan, and Tracy Hall. "De-motivators for software process improvement: an analysis of practitioners' views." Journal of Systems and Software 66.1 (2003): 23-33.

[39] Hindriks, Koen V. "Debugging is explaining." PRIMA 2012: Principles and Practice of Multi-Agent Systems. Springer Berlin Heidelberg, 2012. 31-45.

A. ANTLR Grammars

A.1. MAS grammar

```
grammar MAS;

mas:
                environment?
                agentFiles
                launchPolicy
                EOF
            ;
// ENVIRONMENT
environment:
                ENVSECTION CLBR
                    environmentFile
                    (INIT EQUALS SLBR initParams SRBR DOT)?
                CRBR
            ;
environmentFile:
                ENV EQUALS DOUBLESTRING DOT
            ;
initParams:
                initParam (COMMA initParam)*
            ;
initParam:
                ID EQUALS initValue
            ;
initValues:
                initValue (COMMA initValue)*
            ;
initValue:
                simpleInitValue | functionInitValue | listInitValue
            ;
simpleInitValue:
                ID | INT | FLOAT | SINGLESTRING | DOUBLESTRING
            ;
```

```
functionInitValue:
                ID LBR initValues RBR
            ;
listInitValue:
                SLBR initValues SRBR
            ;
// AGENTFILES
agentFiles:
                AGENTSECTION CLBR
                    agentFile*
                CRBR
            ;
agentFile:
                DOUBLESTRING (SLBR agentFileParameters SRBR)? DOT
            ;
agentFileParameters:
                agentFileParameter (COMMA agentFileParameter)*
            ;
agentFileParameter:
                (NAME | LANGUAGE) EQUALS ID
            ;
// LAUNCHPOLICY
launchPolicy:
                LAUNCHSECTION CLBR
                    launchRule*
                CRBR
            ;
launchRule:
                simpleLaunchRule | conditionalLaunchRule
            ;
simpleLaunchRule:
                LAUNCH launchRuleComponents DOT
            ;
launchRuleComponents:
                launchRuleComponent (COMMA launchRuleComponent)*
            ;
launchRuleComponent:
                (((WILDCARD | ID) AGENTFILENAME) | ID) (SLBR INT SRBR)?
            ;
conditionalLaunchRule:
                WHEN entityDescription ATENV DO simpleLaunchRule
            ;
entityDescription:
                ENTITY | (SLBR entityConstraints SRBR)
```

```
            ;
entityConstraints:
                entityConstraint (COMMA entityConstraint)*
            ;
entityConstraint:
                ((NAME | TYPE) EQUALS ID) | (MAX EQUALS INT)
            ;

// ENVIRONMENT TOKENS
ENVSECTION:     'environment';
ENV:            'env';
INIT:           'init';
// AGENT TOKENS
AGENTSECTION:   'agentfiles' ;
NAME:           'name'; // used in launch-section as well
LANGUAGE:       'language';
// LAUNCH TOKENS
LAUNCHSECTION:  'launchpolicy';
WHEN:           'when';
ENTITY:         'entity';
ATENV:          '@env';
DO:             'do';
LAUNCH:         'launch';
TYPE:           'type';
MAX:            'max';
WILDCARD:       '*';
AGENTFILENAME:  ':'[ \t]*~[ \t\f\r\n?%*:|<>.,]+;
// GENERAL TOKENS
ID:             (ALPHA | SCORE) (ALPHA | DIGIT | SCORE)*;
fragment ALPHA: [a-zA-Z];
fragment SCORE: '_';
FLOAT:          (PLUS | MINUS)? (DIGITS DOT DIGITS*) | (DOT DIGITS);
INT:            (PLUS | MINUS)? DIGITS;
fragment DIGITS:DIGIT+;
fragment DIGIT: [0-9];
PLUS:           '+';
MINUS:          '-';
EQUALS:         '=';
DOT:            '.';
COMMA:          ',';
LBR:            '(';
RBR:            ')';
CLBR:           '{';
CRBR:           '}';
```

```
SLBR:           '[';
SRBR:           ']';
SINGLESTRING:   ('\'' ('\\\'' | .)*? '\'');
DOUBLESTRING:   ('"' ('\\"' | .)*? '"');
// SPECIAL TOKENS
LINE_COMMENT:   '%' ~[\r\n]* '\r'? '\n' -> channel(HIDDEN);
BLOCK_COMMENT:  '/*' .*? '*/' -> channel(HIDDEN);
WS:             [ \t\f\r\n]+ -> channel(HIDDEN);
```

A.2. GOAL grammar

```
parser grammar GOALParser;
options{tokenVocab=GOALLexer;}

modules:        (moduleImport | module)+ EOF
            ;

// MAIN
moduleImport:   IMPORT MODULEFILE DOT
            ;

module:         moduleDef (SLBR moduleOptions SRBR)? CLBR
                    knowledge?
                    beliefs?
                    goals?
                    program?
                    actionSpecs?
                CRBR
            ;

moduleDef:      (MODULE function) | (INIT MODULE) |
                (MAIN MODULE) | (EVENT MODULE)
            ;

moduleOptions:  moduleOption (COMMA moduleOption)*
            ;

moduleOption:   exitOption | focusOption
            ;

exitOption:     EXIT EQUALS (ALWAYS | NEVER | NOGOALS | NOACTION)
            ;
```

```
focusOption:    FOCUS EQUALS (NONE | NEW | SELECT | FILTER)
            ;

// KNOWLEDGE, BELIEFS, GOALS
knowledge:      KNOWLEDGE CLBR
                    KR_BLOCK*
                CRBR
            ;

beliefs:        BELIEFS CLBR
                    KR_BLOCK*
                CRBR
            ;

goals:          GOALS CLBR
                    KR_BLOCK*
                CRBR
            ;

// ACTIONSPECS
actionSpecs:    ACTIONSPEC CLBR
                    actionSpec*
                CRBR
            ;

actionSpec:     function (INTERNAL | ENVIRONMENTAL)? CLBR
                    actionPre
                    actionPost
                CRBR
            ;

actionPre:      PRE CLBR KR_BLOCK* CRBR
            ;

actionPost:     POST CLBR KR_BLOCK* CRBR
            ;

function:        ID (LBR KR_STATEMENT+ RBR)?
            ;

// PROGRAM
program:        PROGRAM (SLBR orderOption SRBR)? CLBR
                    macro*
```

```
                programRule*
            CRBR
        ;

macro:      DEFINE function conditions DOT
        ;

orderOption:    ORDER EQUALS
            (LINEAR | LINEARALL | RANDOM | RANDOMALL | ADAPTIVE)
        ;

programRule:    ifRule | forallRule | listallRule
        ;

ifRule:     IF conditions THEN ((actions DOT)|anonModule)
        ;

forallRule:     FORALL conditions DO ((actions DOT)|anonModule)
        ;

listallRule:    LISTALL ((ID RTLARROW conditions) |
            (conditions LTRARROW ID)) DO ((actions DOT)|anonModule)
        ;

conditions:     condition (COMMA condition)*
        ;

condition:      TRUE | mentalRule | (NOT LBR mentalRule RBR)
        ;

mentalRule:     mentalAction | function
        ;

mentalAction:   (selector DOT)? mentalAtom LBR KR_STATEMENT+ RBR
        ;

mentalAtom:     BELIEF | GOAL | AGOAL | GOALA
        ;

actions:    action (PLUS action)*
        ;

action:     (actionAtom LBR KR_STATEMENT+ RBR) | function |
            EXITMODULE | INIT | MAIN | EVENT
```

```
            ;

actionAtom:     (selector DOT)? (ADOPT | DROP | INSERT | DELETE |
                SEND | SENDONCE | LOG | PRINT)
            ;

selector:       selectExp | (SLBR selectExp (COMMA selectExp)? SRBR)
            ;

selectExp:      SELF | ALL | ALLOTHER | SOME | SOMEOTHER | THIS | ID
            ;

anonModule:     CLBR programRule+ CRBR
            ;

lexer grammar GOALLexer;
@members{int bmatch=0,smatch=0;}

// MAIN TOKENS
IMPORT:         '#import';
MODULEFILE:     '"' ~[ \t\f\r\n?%*:|<>]+ '.mod2g"';
MODULE:         'module';
INIT:           'init';
MAIN:           'main';
EVENT:          'event';
FOCUS:          'focus';
NONE:           'none';
NEW:            'new';
FILTER:         'filter';
SELECT:         'select';
EXITMODULE:     'exit-module'; // up here because of the next token
EXIT:           'exit';
ALWAYS:         'always';
NEVER:          'never';
NOGOALS:        'nogoals';
NOACTION:       'noaction';
KNOWLEDGE:      'knowledge' -> pushMode(KRBLOCK);
BELIEFS:        'beliefs'   -> pushMode(KRBLOCK);
GOALS:          'goals'     -> pushMode(KRBLOCK);
// PROGRAM TOKENS
PROGRAM:        'program';
ORDER:          'order';
LINEARALL:      'linearall';
LINEAR:         'linear';
```

```
RANDOMALL:      'randomall';
RANDOM:         'random';
ADAPTIVE:       'adaptive';
DEFINE:         '#define';
IF:             'if';
FORALL:         'forall';
LISTALL:        'listall';
RTLARROW:       '<-';
LTRARROW:       '->';
THEN:           'then';
DO:             'do';
NOT:            'not';
TRUE:           'true';
BELIEF:         'bel'       -> pushMode(KRSTATEMENT);
AGOAL:          'a-goal'    -> pushMode(KRSTATEMENT);
GOALA:          'goal-a'    -> pushMode(KRSTATEMENT);
GOAL:           'goal'      -> pushMode(KRSTATEMENT);
ADOPT:          'adopt'     -> pushMode(KRSTATEMENT);
DROP:           'drop'      -> pushMode(KRSTATEMENT);
INSERT:         'insert'    -> pushMode(KRSTATEMENT);
DELETE:         'delete'    -> pushMode(KRSTATEMENT);
LOG:            'log'       -> pushMode(KRSTATEMENT);
PRINT:          'print'     -> pushMode(KRSTATEMENT);
SENDONCE:       'sendonce'  -> pushMode(KRSTATEMENT);
SEND:           'send'      -> pushMode(KRSTATEMENT);
ALLOTHER:       'allother';
ALL:            'all';
SOMEOTHER:      'someother';
SOME:           'some';
SELF:           'self';
THIS:           'this';
// ACTIONSPEC TOKENS
ACTIONSPEC:     'actionspec';
ENVIRONMENTAL:  '@env';
INTERNAL:       '@int';
PRE:            'pre'       -> pushMode(KRBLOCK);
POST:           'post'      -> pushMode(KRBLOCK);
// GENERAL TOKENS
ID:             (ALPHA | SCORE) (ALPHA | DIGIT | SCORE)*
                { int IDi=1; // 'hack' for KR parameters
                  while(true){
                          final char next = (char)_input.LA(IDi);
                          if(!java.lang.Character.isWhitespace(next)){
                                  if(next=='(') pushMode(KRSTATEMENT);
```

```
                                    break;
                                }
                                IDi++;
                            }
                        };
fragment ALPHA: [a-zA-Z];
fragment SCORE: '_';
fragment DIGITS:DIGIT+;
fragment DIGIT: [0-9];
PLUS:           '+';
MINUS:          '-';
EQUALS:         '=';
DOT:            '.';
COMMA:          ',';
LBR:            '(';
RBR:            ')';
CLBR:           '{';
CRBR:           '}';
SLBR:           '[';
SRBR:           ']';
// SPECIAL TOKENS
LINE_COMMENT:   '%' ~[\r\n]* '\r'? '\n' -> channel(HIDDEN);
BLOCK_COMMENT:  '/*' .*? '*/' -> channel(HIDDEN);
WS:             [ \t\f\r\n]+ -> channel(HIDDEN);

mode KRBLOCK;
KR_CLBR:        WS? CLBR
                { setType(CLBR);
                  bmatch++;
                  if(bmatch>1) more();
                };
KR_CRBR:        CRBR WS?
                { setType(CRBR);
                  bmatch--;
                  if(bmatch==0) popMode();
                  else more();
                };
KR_BLOCK:       .;

mode KRSTATEMENT;
KR_LBR:         WS? LBR
                { setType(LBR);
                  smatch++;
```

```
                    if(smatch>1) more();
                  };
KR_RBR:           RBR WS?
                  { setType(RBR);
                    smatch--;
                    if(smatch==0) popMode();
                    else more();
                  };
KR_STATEMENT:     .;
```

B. Evaluation Results

B.1. Exploration

Case Processing Summary

	Which IDE are you using for developing your agent programs?	Cases					
		Valid		Missing		Total	
		N	Percent	N	Percent	N	Percent
Cummulative Score	Eclipse1	64	100,0%	0	0,0%	64	100,0%
	Eclipse2	56	100,0%	0	0,0%	56	100,0%
	Eclipse3	6	100,0%	0	0,0%	6	100,0%
	Simple	53	100,0%	0	0,0%	53	100,0%

Descriptives

	Which IDE are you using for developing your agent programs?		Statistic	Std. Error
Cummulative Score	Eclipse1	Mean	53,281	2,0862
		95% Confidence Interval for Mean — Lower Bound	49,112	
		95% Confidence Interval for Mean — Upper Bound	57,450	
		5% Trimmed Mean	53,672	
		Median	55,000	
		Variance	278,547	
		Std. Deviation	16,6897	
		Minimum	12,5	
		Maximum	82,5	
		Range	70,0	
		Interquartile Range	22,5	
		Skewness	-,301	,299
		Kurtosis	-,278	,590
	Eclipse2	Mean	42,589	2,0455
		95% Confidence Interval for Mean — Lower Bound	38,490	
		95% Confidence Interval for Mean — Upper Bound	46,689	
		5% Trimmed Mean	42,887	
		Median	42,500	
		Variance	234,310	
		Std. Deviation	15,3072	
		Minimum	5,0	
		Maximum	75,0	
		Range	70,0	
		Interquartile Range	19,4	
		Skewness	-,361	,319
		Kurtosis	-,059	,628

	Eclipse3	Mean		73,750	3,7500
		95% Confidence Interval for Mean	Lower Bound	64,110	
			Upper Bound	83,390	
		5% Trimmed Mean		73,333	
		Median		73,750	
		Variance		84,375	
		Std. Deviation		9,1856	
		Minimum		65,0	
		Maximum		90,0	
		Range		25,0	
		Interquartile Range		13,8	
		Skewness		1,143	,845
		Kurtosis		1,760	1,741
	Simple	Mean		47,642	2,3129
		95% Confidence Interval for Mean	Lower Bound	43,000	
			Upper Bound	52,283	
		5% Trimmed Mean		47,867	
		Median		45,000	
		Variance		283,513	
		Std. Deviation		16,8379	
		Minimum		7,5	
		Maximum		80,0	
		Range		72,5	
		Interquartile Range		25,0	
		Skewness		-,095	,327
		Kurtosis		-,516	,644

B.2. Reliability

Case Processing Summary			
		N	%
Cases	Valid	179	100,0
	Excluded[a]	0	,0
	Total	179	100,0

a. Listwise deletion based on all variables in the procedure.

Reliability Statistics		
Cronbach's Alpha	Cronbach's Alpha Based on Standardized Items	N of Items
,841	,842	10

Item Statistics			
	Mean	Std. Deviation	N
Frequency	1,58	1,217	179
Complexity	2,28	1,044	179
Ease of Use	2,06	,984	179
Technicality	2,63	1,080	179
Integration	1,72	,984	179
Inconsistency	1,79	1,020	179
Learnability	1,86	1,085	179
Awkwardness	1,91	1,193	179
Confidence	1,70	1,010	179
Start-up Cost	2,06	1,100	179

Summary Item Statistics							
	Mean	Minimum	Maximum	Range	Maximum / Minimum	Variance	N of Items
Item Means	1,958	1,581	2,631	1,050	1,664	,098	10
Item Variances	1,155	,968	1,481	,513	1,530	,032	10
Inter-Item Covariances	,400	,024	,690	,667	29,288	,021	10
Inter-Item Correlations	,348	,022	,517	,495	23,323	,014	10

Item-Total Statistics					
	Scale Mean if Item Deleted	Scale Variance if Item Deleted	Corrected Item-Total Correlation	Squared Multiple Correlation	Cronbach's Alpha if Item Deleted
Frequency	18,00	38,180	,522	,375	,828
Complexity	17,30	40,673	,431	,240	,836
Ease of Use	17,52	38,779	,633	,429	,818
Technicality	16,95	41,722	,331	,215	,845
Integration	17,87	40,802	,456	,366	,833
Inconsistency	17,79	39,056	,581	,400	,822
Learnability	17,72	37,854	,635	,450	,817
Awkwardness	17,68	36,580	,658	,464	,814
Confidence	17,88	38,205	,663	,476	,815
Start-up Cost	17,53	39,723	,474	,283	,832

Scale Statistics			
Mean	Variance	Std. Deviation	N of Items
19,58	47,503	6,892	10