Großer Beleg

Palamedes
A General Game Playing IDE

by

Ingo Keller

Dresden University of Technology
Department of Computer Science
Artificial Intelligence Institute
Computational Logic Group

Supervised by:

Prof. Dr. rer. nat. habil. Michael Thielscher
Dipl.-Inf. Stephan Schiffel

Submitted on September 7, 2009
Keller, Ingo
Palamedes
Großer Beleg, Department of Computer Science
Dresden University of Technology, September 2009
The Game Description Language (GDL) is the language of choice for the annual AAAI General Game Playing Competition and therefore widely adopted by the scientific community. It is used to describe finite, discrete, deterministic multi-player games of complete information. During such a competition the player gets a scrambled description of a game, which he is expected to win without prior knowledge.

Therefore one of the player’s tasks is a syntactic and semantic analysis of the description. To conduct such an analysis, knowledge about general features of game descriptions should be known beforehand.

Thus, there should be a source code editor to support that work. However, up to now there was no specialized tool available. For this task the Palamedes IDE has been developed which combines features of an Integrated Development Environment (IDE) like syntax highlighting with advanced features for GDL analysis like graph views for GDL rules.
Contents

1 Motivation 1

2 Principles of General Game Playing 3
   2.1 Games .................................................. 3
   2.2 Game Model ............................................. 5
   2.3 Game Description Language .......................... 6
   2.4 Analysis .................................................. 8
   2.5 Game Play ................................................ 10
   2.6 Conclusions ............................................. 11

3 Architecture 12
   3.1 Framework Usage .............................. 15
      3.1.1 IDE ................................................ 15
      3.1.2 Library ............................................. 15
      3.1.3 Debugger ........................................... 15
      3.1.4 Simulator .......................................... 16

4 KIF Core plug-in 17
   4.1 The Parser ............................................ 17
   4.2 The Abstract Syntax Tree .......................... 19
   4.3 The Knowledge Base ................................... 20
      4.3.1 IKBFilter ........................................ 21
      4.3.2 IKBAnalyser ...................................... 22

5 GDL Core plug-in 24
   5.1 AST .................................................... 24
   5.2 Knowledge Base ....................................... 25
   5.3 Resolver ................................................ 27
   5.4 Game Model ........................................... 28
   5.5 Simulation ............................................... 30

6 UI plug-in 32
   6.1 Editors ................................................ 32
      6.1.1 Source Code ..................................... 32
      6.1.2 Graph View ..................................... 33
      6.1.3 Game Tree ....................................... 33
      6.1.4 Simulation ....................................... 34
   6.2 Views .................................................. 36
      6.2.1 Outline ........................................... 36
## Contents

6.2.2  Knowledge Base ........................................... 36  
6.2.3  Game State .............................................. 37  
6.2.4  Statistics ................................................. 38  

7  Extensions .................................................. 39  
7.1  Resolver Adaptation ....................................... 39  
7.2  Game Model Interface .................................... 40  
7.3  Strategy Interface ........................................ 41  

8  Conclusions .................................................. 43  

A  Parser Definition ............................................ 45  

B  Example Resolver Adaptation with JavaProver ............ 48  

Bibliography ................................................... 51
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Game Play</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Game State Representation</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Counter Graph</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Player Agent</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Palamedes plug-in Architecture</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Abstract Syntax Tree UML</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Knowledge Base</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Simulator Overview</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>Source Code Editor with Outline &amp; Knowledge Base View</td>
<td>33</td>
</tr>
<tr>
<td>6.2</td>
<td>Game Tree &amp; Game State</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Simulation Editor</td>
<td>35</td>
</tr>
<tr>
<td>6.4</td>
<td>Indicator Symbol Overview</td>
<td>36</td>
</tr>
<tr>
<td>7.1</td>
<td>Strategy Interface</td>
<td>41</td>
</tr>
</tbody>
</table>
## List of Tables

2.1  GDL predefined predicates ........................................ 7
3.1  Package Structure Overview ....................................... 14
4.1  Default keys for the knowledge base entries ................... 20
4.2  Determined order types ........................................... 22
5.1  Additional GDL AST Elements ..................................... 24
5.2  GDLForm Helper Methods .......................................... 25
5.3  Game Model Interfaces ............................................. 28
5.4  Access to Game Components ....................................... 29
5.5  Example Strategy Implementations ................................. 31
7.1  Game Model Interface Implementations ........................... 40
Listings

2.1 Partial Game Description: 8-Puzzle ............................. 8
2.2 Counter Pattern ................................................. 9
4.1 KIF Grammar ..................................................... 17
4.2 kif.core.GroundChecker.java ................................. 19
4.3 kif.core.knowledgebase.filter.FunctionChecker .......... 22
7.1 Register a strategy implementation ......................... 42
A.1 Scanner Definition (KIF.flex) ............................... 45
A.2 Parser Definition (KIF.cup) ................................. 46
B.1 gdl.resolver.javaprover.FluentAdapter.java .............. 48
B.2 gdl.resolver.javaprover.TermAdapter.java .............. 48
B.3 gdl.resolver.javaprover.StateAdapter.java .............. 48
B.4 gdl.resolver.javaprover.GameJP.java .................... 49
B.5 gdl.resolver.javaprover.ReasonerAdapter.java .......... 49
1 Motivation

Since ancient times the human mind likes to play games to amuse itself in winning a prize, showing off someone’s superiority or just for fun. Some of these games are games of luck like throwing dices in a casino, others combine the element of chance with a plan how to get ahead of the concurrent players as in poker. Then there are games that are totally targeted at the strategic ability of the playing parties like Chess for example. Compared to the long history of game playing only recently the human mind got an alien opponent, the computer. While the first Chess playing automaton “The Turk” still was a fake, made for amusement and played by a hidden person, the development of automatons and later on computers has lead to such complex devices as the Hydra machine, an amazingly sophisticated machine, which was solely constructed for the purpose of playing Chess.

Being tempting in the case of big jackpots, games of chance don’t have much to offer for the adept of game playing. Much more is offered by strategy based games since strategies can be learned and some of them even fill a vast amount of books as it is the case with Chess. However, the methods used to approach computer based game playing are as diverse as the games themselves. To get a chance of learning a strategy to win a game or even solve it, one has to restrict the class of games under research. In the past, restricting to a game class narrowed the approach down to a single game. This resulted in highly specific algorithms that mostly represented the programmers knowledge about the game in question. Only recently General Game Playing (GGP) entered the stage with the aim to develop algorithms that can be applied to a full range of games by an agent that doesn’t even have to know beforehand, which game it will be playing.

Probably the main driving force in the GGP area is the AAAI Competition [2]. Once a year, this international competition takes place to promote the development of new methods and algorithms. A platform for comparing different approaches was originated and had established a research framework. Becoming aware of the Fluxplayer [10] winning one of the competitions, the author took a GGP class out of curiosity and soon realized that this area of artificial intelligence is quite young and therefore provides a wide range of research opportunities. Being instructed to develop an own agent his team found that some tools are missing or are not appropriate for the task. By implementing and comparing different strategies, we found ourselves developing the same basic player as a lot of people did before. But due to the lack of documentation, incompatible systems or just broken implementations
Chapter 1 Motivation

none of those proved applicable and we had to start from scratch nevertheless. Un-
satisfied with this outcome we wanted to provide a commonly usable framework for
developing own agents but at the same time we would like to offer more than just
another tutorial basic player. Said that, we also should point out that we missed de-
velopmental tools as well, things like an Integrated Development Environment (IDE)
or contextual visualisation.

All these smaller and bigger problems have lead us to implementing the Palamedes
IDE (or short Palamedes). We believe that Palamedes is able to fill in this position
and is a potent tool against these shortcomings. This paper will show how this is
achieved and how the IDE can be used further to create own applications. However,
even though the IDE is able to cope with real world tasks, it’s main focus is still
on the teaching aspect. Therefore the implementation prefers a clear and under-
standable source code over an optimised one, sometimes resulting in a certain lack
of performance.

The remainder of this paper is organised as follows. Chapter 2 describes the back-
ground on which Palamedes IDE operates. After giving the overview about certain
aspects we talk about the project architecture in chapter 3. This will show the un-
derlying design aspects and how the parts of the application work together. Chapter
4 and 5 will give a detailed overview about the basic plug-ins. In chapter 6 the usage
of the user interface plug-in will be discussed. Before we will draw a conclusion we
give some ideas of how to extend the Palamedes IDE for using it in own projects in
chapter 7. Finally chapter 8 gives some ideas that already have arisen about how to
develop the IDE even further.
2 Principles of General Game Playing

It is estimated that the game of Chess has between $10^{43}$ to $10^{50}$ game states \[1\]. Solving the game by looking at each game state, and therefore in a brute-force manner, is quite impossible with the technology at hand. Even all supercomputer on earth, currently having a combined power of more than 22 PetaFLOPS\[1\], would have to run for longer than the life time of the universe to perform such an evaluation. Nevertheless, as already mentioned, there is a lot research done to solve Chess for example by applying a wide variety of heuristic based search algorithms. Today there are Chess computers available that can evaluate game states a couple of levels deep from a starting position. However, the evaluation method itself contains Chess knowledge from a large number of human minds and therefore one could hesitate to call it an artificial intelligence approach. And most importantly this knowledge can not be applied to a different game. In the following sections we want to show the principles of General Game Playing and how they are used to approach the problems mentioned above. This is considered as an overview of the topic referring to other work for further details.

2.1 Games

Since games are quite diverse in their nature and come with all sorts of twists that are hard to grasp formally or computationally, GGP has restricted the class of games to be able to cope with them. As outlined by \[2\] the class of games under consideration is defined as finite and synchronous. The finiteness property demands that games take place in an environment with a finite number of states, comprising one distinguished initial state and terminal states with associated goal values. Additionally every player has only a finite number of possible actions in each state and there is only a finite number of players involved. A synchronous game is a game in which all players move at once and on all steps, although moves can be skipped doing a “no-op” action which is needed for turn-based games. The environment only changes in response to player’s moves.

\[5\] introduces the requirement of the deterministic and perfect information properties. The first demands that each unique combination of a game state and a joint set of actions by all players lead deterministically to an unique successor

\[1\] 22.6 * 10^{15} FLOPS in June 2009 (http://www.top500.org - TOP 500 Supercomputer list)
game state. While interesting games like Chess, Checkers and Go are showing this property all games involving an element of chance are prohibited especially all kind of card games like poker or games involving rolling dices like Backgammon. Another feature that most card games do not show is the perfect information property. In perfect information games all game elements and their state is known and visible at every time to all participants. Apparently this is not the case if the game only can be played wisely if information is kept secret.

Although the range of games now seems a bit small in the first place, even simple ones can exceed the capability of modern computers. While some of them (e.g. Checkers [8]) are completely solved already, the more complex ones are not solvable by iterating over every game state, therefore other approaches have to be taken. To be able to compare completely different solutions the AAAI competition was created. This competition is meant to be a platform for students and researchers to share their knowledge and ideas and to test and compare their algorithm implementations on real games. Therefor a standardized communication protocol and a game master server have been provided. For each competition a set of games was created that the players didn’t know beforehand. The games were specially written to raise the complexity level from year to year. Figure 2.1 shows the interaction of the available software components. The player boxes on the bottom denote for the implementations the teams have to provide.

To play a game, the game master server sends the description to player agents. In this description the goals for each player, their legal moves, the initial state and
2.2 Game Model

Although, the games at hand can have different properties, like single player vs. multi-player or turn taking vs. synchronous moves, they can be formalised in the same way. The mentioned properties can be seen as behavior emerging from the rules. Whereas most games currently under inspection fall in only one of these classes in principle it is possible to think about a joint game that shows all of this properties at once. Therefore a general game model is necessary.

As such a \textit{n-player game} can be formally represented as shown in Definition 2.1. It can be seen as a state machine (see Figure 2.2), which starts in state $s_0$. The player is called \textit{role} and on each step, each role $r_i$ chooses an \textit{action} $l(r_i, s_j) \in A$ which is legal in the current state $s_i$. The game then progresses to the next state $s_j$ using the update function. This continues until the game reaches a state in which $s \in t$ holds and therefore ends. With this terminal state and the goal values the outcome for each player can be resolved. The role with the highest outcome wins and equal goal values are considered a draw.
Definition 2.1. **Game Components**

\[ R, \] the set of roles, with \(|R| = n\),
\[ S, \] the set of states,
\[ A, \] the set of actions,
\[ l : R \times S \rightarrow A \] the legality relation,
\[ u : P(R \times A) \times S \rightarrow S \] the update function
\[ s_0 \in S \] the initial state,
\[ t \subseteq S, \] the terminal states, and
\[ g : R \times S \rightarrow N, \] the goal values \( g_i \in [0, \ldots, 100] \).

### 2.3 Game Description Language

As already stated, the AAAI competition is the boundary framework for the development of General Game Playing. To ease communication between all the competitors the founder of the competition introduced an own language. Developed in 2005 at the Stanford University and updated in 2008 the **Game Description Language** (GDL) is used to describe games for GGP [7]. It is a specialisation of the **Knowledge Interchange Format** (KIF), which “is a formal language for the interchange of knowledge among disparate computer programs” [3]. Being a variant of first order language it comprises some relations to support the conceptualization of games. Beyond this no further knowledge like an algebra or arithmetic is required since GDL is purely axiomatic. Every additional game relevant construct has to be defined within the game description.

**Definition 2.2. GDL Syntax**

A **Vocabulary** consists of

- a set of relation symbols with associated arity,
- a set of function symbols with associated arity,
- a set of object constants.

A **Term** is

- a variable, being a symbol with leading ?,
- an object constant,
- a function symbol of arity \( n \) applied to \( n \) terms.

An **Atomic Sentence** is

- a relation symbol of arity \( n \) applied to \( n \) terms.

A **Literal** is

- an atomic sentence or the negation of an atomic sentence.

An expression is **ground** if and only if it contains no variables.

A **Rule** is

- an implication of the form \( h \Leftarrow b_1 \land \ldots b_n \).

The head \( h \) is an atomic sentence.

Each literal in the body \( b_i \) is a literal or a disjunction of literals.
### 2.3 Game Description Language

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role</td>
<td>defines the role names</td>
</tr>
<tr>
<td>Init</td>
<td>defines the initial state</td>
</tr>
<tr>
<td>Does</td>
<td>indicates the moves made by players in one step</td>
</tr>
<tr>
<td>True</td>
<td>defines the facts that hold in the current state</td>
</tr>
<tr>
<td>Next</td>
<td>defines the facts that hold in the next state</td>
</tr>
<tr>
<td>Legal</td>
<td>indicates the legal actions of the roles</td>
</tr>
<tr>
<td>Terminal</td>
<td>defines the game state terminal conditions</td>
</tr>
<tr>
<td>Goal</td>
<td>defines the game state goal values</td>
</tr>
</tbody>
</table>

Table 2.1: GDL predefined predicates

However, some conditions have to be met by the description to produce **well-formed** games [7].

- **Termination** A game described by GDL terminates if all infinite sequences of legal moves from the initial state of the game can reach a terminal state after a finite number of steps.

- **Playability** A game described by GDL is playable if and only if every role has at least one legal move in every non-terminal state reachable from the initial state.

- **Winnability** A game described by GDL is strongly winnable if and only if, for some role, there is a sequence of individual moves of that role that leads to a terminal state of the game where that role’s goal value is maximal. A game described by GDL is weakly winnable if and only if, for every role, there is a sequence of joint moves of all roles that leads to a terminal state where that role’s goal value is maximal.

- **Well-formed Games** A game described by GDL is well-formed if it terminates and is both playable and weakly winnable.

Additionally the GDL comes with a small set of predefined predicates that are the basis for every game shown in Table 2.1. An example of such a game description is given in Listing 2.1. It shows a part of the game 8-Puzzle. In such a GDL description, games are modeled as state machines in which a state is the set of true facts at a given time. An agent can derive its legal moves, the next state given the moves of all players, and whether or not it has won by applying resolution theorem proving.
Chapter 2 Principles of General Game Playing

2.4 Analysis

The Game Description Language is used to communicate the rules of the game. However, a player agent will translate this representation into an internal one for the actual resolving process. In this transformation normally an analysis of the source code is done. Every game description contains some intrinsic information about the rules and game structures. As pointed out this information should be extracted to facilitate the process of rule optimisation or even strategy selection.

Given a reasoner and arguing that improvements to it are not necessarily part of the GGP approach, one can distinguish three different levels of analysis an agent can do to optimise its overall performance. In the first step the source code is read resulting in an Abstract Syntax Tree (AST), which can be transformed to speed up the reasoning procedure. Transformations like expanding simple rules or rule reordering can be applied in order to improve the performance of the resolver. Nonetheless this is done purely on the syntactic level. In a next step the agent player can try
2.4 Analysis

to determine game structures and therefore try to find the semantics behind game elements. This is done to identify constructs like step counters (see Listing 2.1 line 11-13) and board pieces [5] as well as for game properties like board symmetry [9]. The last pivotal point is the choice of the strategy for the played game. This can be one overall strategy as seen in the Monte Carlo derived approaches [4] but of course the strategy could also be chosen upon knowing the properties of the particular game. Especially games that require a vast amount of time to resolve the legal moves or the next states, and thus are hard to simulate up-front, could gain from knowing the game internals.

Listing 2.2: Counter Pattern

<table>
<thead>
<tr>
<th></th>
<th>&lt;= (next (&lt;counter&gt; ?&lt;var1&gt;))</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(true (&lt;counter&gt; ?&lt;var2&gt;))</td>
</tr>
<tr>
<td>3</td>
<td>(&lt;successor&gt; ?&lt;var2&gt; ?&lt;var1&gt;))</td>
</tr>
<tr>
<td>4</td>
<td>(&lt;successor&gt; 1 2)</td>
</tr>
<tr>
<td>5</td>
<td>(&lt;successor&gt; 2 3)</td>
</tr>
</tbody>
</table>

As showcase Figure 2.3 shows a graph which can be conducted from syntactic constructs like Listing 2.2. The ellipses in the figure denote arguments of predicates and the squares denote constant symbols. After an automated creation such a graph can be matched and therefore recognized. However that is not possible for every kind of construct so far. [6]

Some of the constructs are reoccurring, that means they are included in several games and therefore can be called a game element. The example above shows a general counter. A common instance of it is the step counter as shown earlier. With the knowledge of the existence of that construct and its behavior it might be able to remove it from the intermediate game calculation resulting in a better performance.
2.5 Game Play

To actually play a game a player agent is needed. Figure 2.4 shows an example of such an agent. Given that the agent has already constructed its internal representation of the game as outlined above the agent still needs a way to use the game model to find best moves. This is done by utilizing different search algorithms. The searches (here also called strategies) are responsible for the action selection based on the current state of the game. Whereas the game is theoretically represented by a game state graph which is a directed graph that can contain cycles, most of the basic agent implementations use an acyclic game state tree as representation for their searches. Some approaches avoid the cycles by keeping hashtables of already resolved nodes to find cycles within the tree. However this comes with some overhead for the bookkeeping of the states and might not be possible in all cases.

The very elementary searches that an agent might use can be found in every textbook. The **depth-first search** (DFS) expands the game state tree by expanding the deepest node first. This can be done with the lowest memory footprint of all searches but comes at the cost of not being able to deal with cycles and the shortest path won’t be found this way either. On the opposite side is the **breadth-first**
search (BFS). It expands the tree level by level and is therefore able to find the shortest path between the root node and the solution. The drawback is that it needs to keep a growing part of the tree in the memory and if the branching factor of the tree is high (many possible moves in a game state) it might even hit the memory boundaries before it finds a terminal state. A combination of these searches can be found in the iterative deepening search (IDS). It shares the strategy of expanding the deepest node first with the DFS but it stops going further if the node that needs to be expanded is deeper than a prespecified depth limit. In the result the search then traverses the whole tree up to that limit. Once the whole subtree is analyzed and no solution has been found so far the limit is raised. This way the search can go deeper than in the first run. With small limits it turns into a breadth-first search and therefore comprises the same properties.

However, all mentioned search strategies fall under the category of uninformed searches. Uninformed searches look for a solution without taking the actual node properties into account. Whereas their ability is sufficient for small games they are clearly not helping if it comes to more complex problems. Therefore more specialised search strategies are developed mostly by utilizing the information of the game states to give the search some kind of heuristic that helps finding the goal state.

### 2.6 Conclusions

In this chapter we had a short overview about the GGP components and how they interact. In the next chapters we will show in detail how the Palamedes framework covers all these aspects and how it can help developing general game playing agents.

Before a game can be played it has to be written. Here the Palamedes IDE starts its work. Having a source code editor, it supports the source code writing with standard IDE features like syntax highlighting, outline views and so on. It is able to show all kind of aspects of the source code from a knowledge base up to simulating the game described by the source. The following section will give an overview about the architecture of Palamedes.
3 Architecture

Providing a development environment, which is simple to use, easily extendable by students and usable for teaching is the main motivation behind the Palamedes IDE. Since Java is widely adopted for computer language teaching, it is safe to assume that this language is a common one among students. Therefore Java was the language of choice even though many of the actual programs in General Game Playing run on Prolog.

With Eclipse an Open-Source Framework exists that is widely used for developing Java applications. It is probably best known for its use as a Java IDE. However, this framework is more than just an IDE. It ships with a lot of subframeworks for different purposes such as support for all kinds of programming languages (C++, Python, Ruby, etc.), modeling frameworks (EMF, GMF, UML2m, etc.) and productivity tools (CVS, SVN, Database Browser, etc.). On top of all these frameworks and tools that come along as Eclipse plug-ins, Eclipse provides a Rich Client Platform (RCP) for developing complete, independent applications. This full range support is possible, because Eclipse strongly relies on its plug-in architecture. Therefore, even the most basic application is a bundle of plug-ins that work together.

The arguments above have led to the decision to use the Eclipse Platform as the environment for the Palamedes IDE since a lot of aspects we were looking for are already provided by Eclipse. First of all we can benefit from all tools available for the Java IDE. Individual source code files can be grouped in projects and file extensions (aka. mime-types) can be bound to specific editor components. The basic text editor already supports operations like copy&paste, replace and undo. Repository support via CVS comes out of the box and the current standard Subversion can be easily installed via a plug-in. This way, even without adding anything specific, Eclipse already can be used as source code editor. Nonetheless, it lacks all language specific GDL support and therefore can not be considered an Integrated Development Environment (IDE). The language specific functionality is added by Palamedes plug-ins to turn the editing support into a full-fledged IDE. Another point that should be stressed here is the fact that the IDE serves as a backbone framework. This means that it has hooks that can be filled with own implementations, as will be shown later in chapter 7.

To facilitate the work on Palamedes IDE, a sourceforge project was created. Access to the source code of the project can be obtained via CVS. The repository
was made publicly available since we believe that the open source software approach is the best for scientific research. A binary distribution that makes use of the Eclipse update manager is available as well and can be used to instantly start using the Palamedes IDE. The project can be found at http://palamedes-ide.sourceforge.net/.

The Palamedes IDE consists of three Eclipse plug-ins as shown in Figure 3.1. The arrows in the figure indicate the information flow between subsystems of each plug-in as well as the information flow towards the next layer. It is shown that the information is transferred from the very basic plug-in to the layers below it. Having three different plug-ins that work on top of each other brings the advantage that each for itself is a separate part of the system with only a few interface where information needs to be handed over. Therefore changing components within one plug-in can be done without affecting the other plug-ins. This is valuable if one wants to debug or analyse the behaviour of components during the optimisation phase of a player agent development. It can also support the teaching of such systems since one can pick out certain components to discuss them in more detail.
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.eclipse.palamedes.kif.core</td>
<td>KIF Core Root Package</td>
</tr>
<tr>
<td>org.eclipse.palamedes.kif.core.ast</td>
<td>KIF Abstract Syntax Tree</td>
</tr>
<tr>
<td>org.eclipse.palamedes.kif.core.knowledgebase</td>
<td>KIF Knowledge Base</td>
</tr>
<tr>
<td>org.eclipse.palamedes.kif.core.parser</td>
<td>KIF Parser</td>
</tr>
<tr>
<td>org.eclipse.palamedes.gdl.core</td>
<td>GDL Core Root Package</td>
</tr>
<tr>
<td>org.eclipse.palamedes.gdl.core.ast</td>
<td>GDL Abstract Syntax Tree</td>
</tr>
<tr>
<td>org.eclipse.palamedes.gdl.core.knowledgebase</td>
<td>GDL Knowledge Base</td>
</tr>
<tr>
<td>org.eclipse.palamedes.gdl.core.model</td>
<td>Game Model Part</td>
</tr>
<tr>
<td>org.eclipse.palamedes.gdl.core.resolver</td>
<td>Resolver Part</td>
</tr>
<tr>
<td>org.eclipse.palamedes.gdl.core.simulation</td>
<td>Simulation Part</td>
</tr>
<tr>
<td>org.eclipse.palamedes.gdl.core.connection</td>
<td>Connectivity Part</td>
</tr>
<tr>
<td>org.eclipse.palamedes.ui</td>
<td>UI Root Package</td>
</tr>
<tr>
<td>org.eclipse.palamedes.ui.actions</td>
<td>Action Button Functions</td>
</tr>
<tr>
<td>org.eclipse.palamedes.ui.editor</td>
<td>Editor Parts</td>
</tr>
<tr>
<td>org.eclipse.palamedes.ui.pages</td>
<td>View Content Pages</td>
</tr>
<tr>
<td>org.eclipse.palamedes.ui.preferences</td>
<td>plug-in Preference Section</td>
</tr>
<tr>
<td>org.eclipse.palamedes.ui.views</td>
<td>Own View Implementations</td>
</tr>
</tbody>
</table>

Table 3.1: Package Structure Overview

**KIF Core** is the most basic plug-in. It provides the parser, the abstract syntax tree and a basic knowledge base. The KIF Core gets used by the **GDL Core plug-in**. This one provides everything related to GDL analysis, game modelling and simulations. On top of both of these plug-ins sits the **UI plug-in**, whose components follow the Model View Controller concept. It provides the user interfaces to the GDL Core and contains the source code editor.

Table 3.1 shows how the plug-ins are broken into smaller pieces. The following sections will not use the full package name. Instead we will use *kif*... for the KIF Core plug-in, *gdl*... for the GDL Core plug-in and *ui*... for the UI plug-in. That means `org.eclipse.palamedes.gdl.core.knowledgebase` would be denoted as `gdl.core.knowledgebase` to ease readability.
3.1 Framework Usage

Palamedes IDE was written with several aims in mind. This section will give an overview on how it can be used to assist in different development tasks.

3.1.1 IDE

First of all and according to the name, Palamedes IDE can be used as an Integrated Desktop Environment. It supports syntax highlighting for the Game Description Language together with an outline view that shows the syntactic elements. Both tools can accelerate the writing of description files since the outline view gives an indication whether the source code is syntactically correct and errors can be noticed more easily. The provided knowledge base view gives information about the specific symbols and their properties. This helps to see whether the written source code gets translated as intended or if there are semantic errors in it which can’t be found by the parser.

Eclipse comes with an integrated project management feature that can be used to bundle description files into a project. Together with a versioning system plug-in these projects can be shared and developed by a team of developers, which makes it easier to track errors and facilitate the teamwork.

3.1.2 Library

The plug-ins can be used as a framework for developing your own player agents. Only the UI plug-in is Eclipse dependent but not necessary for writing own agents. The other two plug-ins can be used as a library which contains all tools necessary for an agent. It supports the parsing of the description, the creation of a game model and the communication with the game master. Automatic knowledge base creation and some tools for analyzing it are available. A variety of resolvers can be utilized to find new states and the next moves that the player agent has to do. An interface is provided implementing different strategies which is easy to apply and for which a couple of example implementations are given.

3.1.3 Debugger

For the task of debugging a game the game state tree is provided. It can be used to expand the tree by hand to check if the provided rules work as supposed. This can be complemented by the game state view which is able to show the selected state. With
the help of stylesheets it can visualize certain states of game components like boards and their corresponding pieces. This kind of visualization is an important aspect in finding game errors since failures can be seen directly and the developer does not have to bother with reading the game state information in a text version. At present time the stylesheets still have to be written for each game individually and therefor require an extra amount of work. However, in the future a semi-automatic visualization could be implemented to ease the development of stylesheets.

### 3.1.4 Simulator

Last but not least Palamedes can be used for developing and testing new search strategies. As mentioned above it comes with an extendable strategy interface. Own implementations can be run against each other to see which one is superior. The benefit is that no external game master is required and because communication overhead is avoided the simulations run faster. Intermediate steps and game states can be viewed with the simulation editor which helps to debug the strategies. Some internal parameters are logged by the simulator which can be compared across strategy implementations.
4 KIF Core plug-in

After discussing the general aspects of Palamedes IDE we want to have a closer look at the individual plug-ins. As already mentioned the KIF Core part of the IDE is responsible for everything related to source code handling. Because GDL is a subset of KIF we start with a clean KIF environment and will build the GDL related functionality on top of it.

4.1 The Parser

```
I. Forms
<form> ::= <sentence>*

II. Sentences
<sentence> ::= <relsent> | <logsent>
<relsent> ::= (<relnconst> <term>*)
<logsent> ::= (not <sentence>) | (and <sentence> <sentence>*) | (or <sentence> <sentence>*) | (<= <sentence> <sentence>*)

III. Terms
<term> ::= <indvar> | <objconst> | <funterm>
<funterm> ::= (<funconst> <term>*)

IV. Lexemes
<objconst> ::= <constant>
<funconst> ::= <constant>
<relnconst> ::= <constant>
<logsconst> ::= <constant>

V. Sequence Helper
<sentenceseq> ::= <sentence>+
<termseq> ::= <term>+
```

Listing 4.1: KIF Grammar

Everything starts with the parser. It creates the abstract syntax tree (AST), which is our model of the source code file and will be the starting point for all kinds of analysis and transformations later on. Since GDL does not make use of all aspects of KIF the necessary subset was implemented. This relevant subset is shown in Listing 4.1.
The parser was implemented using the LA LR parser generator CUP\(^1\) in conjunction with the JFlex lexical analyser implementation. JFlex is fed with the lexical description as outlined in Listing A.1. It then generates the `KIFScanner` class, which is used by CUP. CUP itself gets the description of the grammar and generates the corresponding `KIFParser` and `KIFSymbol` classes. Besides generating the AST the parser also delivers the positions of the AST elements which is needed for the source code editing aspect.

For convenience the classes are wrapped by the `Parser` class. This one is the only public class of the Parser component and can be used for parsing a given file (`parseFile` method) or a GDL string (`parseGDL` method). Both methods return either a valid AST object containing the abstract syntax tree elements or they raise an exception that can be caught to handle a failure. However, to keep the interface clean the usual parser calling is done by the `AST` class as we will see in the next section.

\(^1\)\url{http://www2.cs.tum.edu/projects/cup/}
4.2 The Abstract Syntax Tree

Constructed by the parser the abstract syntax tree represents the logical structure of the game description. The AST follows the composition pattern. Figure 4.1 shows the AST nodes of the tree for the KIF language part that got implemented for Palamedes. The parser can map each language element to a corresponding ASTNode type with the help of the AST class which acts as a factory. The factory methods of the AST class are named according to the type of node it creates, e.g. AST.newImplication creates a KIFImplication node. The factory is responsible for creating a valid tree while the parser is feeding it with the corresponding factory method calls. After completion the AST object contains a valid abstract syntax tree constructed of ASTNode objects.

The syntax tree can be either constructed by hand using the factory methods or gets created by invoking either the parseFile or parseString method on an AST object. Once that is done it can be used to traverse over it or to be modified in any way. For traversal the individual nodes contain a traverse method to which a ASTVisitor object is handed over and take care of the invocation of the same method on their children, if there are any. The visitor can gather information about the nodes by calling getNodeType to determine the type of node he is currently visiting, getSourceStart or getSourceLength to get the location within the original source string or individual node methods to get more specific node content like getName in the case of a KIFIndVar node. This way the abstract syntax tree can be inspected for syntactic and semantic information.

```
package org.eclipse.palamedes.kif.core.ast.visitor;
import org.eclipse.palamedes.kif.core.ast.ASTNode;
import org.eclipse.palamedes.kif.core.ast.KIFIndVar;

public class GroundChecker extends DefaultVisitor {
    boolean isGround;

    public void visit(KIFIndVar node) {
        isGround = false;
    }

    public boolean isGround(ASTNode node) {
        isGround = true;
        node.traverse(this);
        return isGround;
    }
}
```

Listing 4.2: kif.core.GroundChecker.java

The base visitor class DefaultVisitor is an implementation of ASTVisitor and can be used to traverse the AST in a depth first manner. It keeps track if a node is inside the body or the head of an implication or not in an implication at all.
Additionally it has a switch (\texttt{HEADFIRST}) to traverse the implication head first or not. For each \texttt{ASTNode} subclass a hook method \texttt{visit} can be implemented to steer the behavior of the visitor if it hits the specific node type.

The example in Listing 4.2 shows an application of such a visitor for inspecting a node and its subtree to check whether it contains a variable. \texttt{ISGROUND} gets a node to start the traversal (line 15), which is done by the \texttt{DefaultVisitor} in the background. Every time a \texttt{KIFINDVAR} is hit, the corresponding \texttt{visit} method is called (line 9). In this example it is used to track the appearance of such an event. The example demonstrates how the syntax tree can be inspected. In the same way changes can be made to the syntax tree in the case it needs to be transformed.

### 4.3 The Knowledge Base

The knowledge base is basically a storage of information about occurring symbols. Its entry class \texttt{KBEntry} is a hash map wrapper, which can store every kind of information about a symbol. For extensibility reasons the key is an arbitrary string and the value is an object. This way the entries can store information not already build into the system.

<table>
<thead>
<tr>
<th>\texttt{KBEntry Constants}</th>
<th>Key String</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_NAME</td>
<td>“Name”</td>
<td>lowercase symbol name</td>
</tr>
<tr>
<td>P_TYPE</td>
<td>“Type”</td>
<td>“Relation” or “Function”</td>
</tr>
<tr>
<td>P_ARITY</td>
<td>“Arity”</td>
<td>arity of the symbol</td>
</tr>
<tr>
<td>P_TOP_LEVEL_NODES</td>
<td>“Top Level Nodes”</td>
<td>top level nodes</td>
</tr>
<tr>
<td>P_IMP_HEAD</td>
<td>“Implication Head”</td>
<td>implication nodes, in which the symbol appears in the head</td>
</tr>
<tr>
<td>P_IMP_BODY</td>
<td>“Implication Body”</td>
<td>implication nodes, in which the symbol appears in the body</td>
</tr>
<tr>
<td>P_NODES</td>
<td>“Node List”</td>
<td>all ASTNodes for this symbol</td>
</tr>
</tbody>
</table>

Table 4.1: Default keys for the knowledge base entries

Table 4.1 shows the default keys which are available in the \texttt{KBEntry} class. They can be inserted by the method \texttt{setProperty} and retrieved by the method \texttt{getProperty}. The \texttt{hasProperty} method allows to check if a property exists without retrieving it. As shown in Figure 4.2 the \texttt{KBEntry} has some more methods to retrieve the associated syntax tree nodes. These are mainly convenience methods for analyzing the individual source code snippets without having to search for them in the AST.
4.3 The Knowledge Base

4.3.1 IKBFilter

All storing of information would be useless if we did not provide a query method. Besides standard get methods the knowledge base contains the filter method, which accepts an IKBFilter query. Implementing this interface just means to create a class, which has a method that accepts a KBEntry and returns a boolean value if this entry satisfies the filter. In Listing 4.3 an example function filter is demonstrated. The function filter only matches symbols that were recognized as function. Filter can be stacked with the provided AndFilter and OrFilter classes. This way complex filter pattern can be constructed and used to match certain symbol properties. Within Palamedes this is used for the knowledge base view filter settings for example (see subsection 6.2.2).
### Chapter 4 KIF Core plug-in

<table>
<thead>
<tr>
<th>Order</th>
<th>Relation Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quasi Order</td>
<td>Reflexivity and Transitivity</td>
</tr>
<tr>
<td>Partial Order</td>
<td>Quasi Order and Antisymmetry</td>
</tr>
<tr>
<td>Total Order</td>
<td>Partial Order and Linearity</td>
</tr>
<tr>
<td>Strict Order</td>
<td>Irreflexivity and Transitivity</td>
</tr>
</tbody>
</table>

Table 4.2: Determined order types

```java
package org.eclipse.palamedes.kif.core.knowledgebase.filter;

import org.eclipse.palamedes.kif.core.knowledgebase.IKBFilter;
import org.eclipse.palamedes.kif.core.knowledgebase.KBEntry;

public final class FunctionFilter implements IKBFilter {
    public boolean isMatching(KBEntry entry) {
        return entry.get(KBEntry.P_TYPE).equals(KBEntry.V_FUN);
    }
}
```

Listing 4.3: kif.core.knowledgebase.filter.FunctionChecker

### 4.3.2 IKBAnalyser

A more sophisticated access to the internals of the knowledge base is given through the use of `IKBAnalyser`. Whereas `IKBFilter` only selects certain entries by a given boolean function the `IKBAnalyser` implementations are getting full access to the knowledge base. Implementations of that interface need to provide the `analyse` method which takes the `KnowledgeBase` object as an argument and acts as the main method for the analysis. For a better recognition the names of classes that implement the interface `IKBAnalyser` start with an “A”.

The very basic information is gathered by `ADomainInformation`, which determines if a symbol is a fact which means that it comprises ground top level nodes. So far no reasoning is done to determine the fact property. Therefore facts within an implication with an empty or fact based body are not recognized. The information is stored in the Property “isFact”. If it has found a fact and the symbol doesn’t have multiple arities it constructs arrays that contain the domain objects by position. This can be retrieved from the Properties “domain_x” in which x is the position. The domain ordering depends on the appearance in the source code. If the analyzer finds a fact but one of the arguments is not an object constant, no domain information is stored. Additionally it stores a distinct set of domain objects
in “domain_distinct”. A possible extension to this analyzer is the reasoning about recursive and complex facts.

**Definition 4.1. Game Components**

- **Reflexivity** \(O(n^2)\) For all elements \(x\) a tuple \((x,x)\) has to be in the relation.
- **Irreflexivity** \(O(n)\) For all elements \(x\) a tuple \((x,x)\) must not be in the relation.
- **Symmetry** \(O(n^2)\) If the relation contains a tuple \((x,y)\) it also has to have a tuple \((y,x)\).
- **Asymmetry** \(O(n^2)\) For all tuple \((x,y)\) a tuple \((y,x)\) must not be in the relation.
- **Antisymmetry** \(O(1)\) The relation is neither symmetric nor asymmetric.
- **Top/Bottom Element** \(O(n)\) Elements that appear only in one position.

For binary facts a separate analysis is done by the **ABINARYFACTS** class. Given the fact is a binary relation it determines the relation properties (ir-)reflexivity, (anti-/a-) symmetry, a start and an end symbol. Definition 4.1 shows the algebraic properties, their definition and the complexity of the implemented tests in the O-notation. From these properties the class derives the type of order the given fact represents as shown in Table 4.2. Due to the complexity of the transitivity test \(O(n^3)\) and its complicated implementation this property is assumed to be given. So far no game description is known that contain binary relations that break this property. However it is clear that one can be written easily. Nonetheless the current implementation serves as an example implementation. Therefore we think it is reasonable to skip this test here and leave it to future implementations.

To be able to conduct the tests the analyzer first retrieves the domain information from the knowledge base. Then it converts the syntax tree nodes into hash codes based on the lower case string representation of constant objects. The conversion needs to be done because the syntax tree nodes could contain different lowercase/uppercase variations of the same constant symbol according to the KIF specification. On the other hand it also speeds up the tests significantly due to the use of integer instead of string comparisons.
5 GDL Core plug-in

Whereas the KIF Core plug-in works syntactically and semantically only on the knowledge of the raw KIF language the GDL plug-in adds all relevant game description features. In order to do this it first adds some more information to the KIF Core syntax tree before doing the game analysis, construction and simulation.

5.1 AST

To reflect the additional language information the KIF based AST gets decorated by GDLAST. It has the same behaviour but overloads the factory methods to create GDL dependent language constructs. The new constructs and their constraints are listed in Table 5.1. Since they inherit from KIF syntax tree nodes (ASTNode) they can be used as such and additionally be queried for their GDL specific information. For example an INIT or TRUE object can be directly asked for the contained argument through the getArgument method whereas the underlying KIFRelSent class would not know about this feature. Therefore this extension can be seen as a convenient way to access the syntax tree using the formalisation of the game description language.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Superclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAMEFORM</td>
<td>KIFForm containing GDL ASTNodes</td>
</tr>
<tr>
<td>DOES</td>
<td>KIFRelSent with does as symbol</td>
</tr>
<tr>
<td>INIT</td>
<td>KIFRelSent with init as symbol</td>
</tr>
<tr>
<td>ROLE</td>
<td>KIFRelSent with role as symbol</td>
</tr>
<tr>
<td>TRUE</td>
<td>KIFRelSent with true as symbol</td>
</tr>
<tr>
<td>RULENEXT</td>
<td>KIFImplication with next as symbol</td>
</tr>
<tr>
<td>RULELEGAL</td>
<td>KIFImplication with legal as symbol</td>
</tr>
<tr>
<td>RuleGoal</td>
<td>KIFImplication with goal as symbol</td>
</tr>
<tr>
<td>RULETERMINAL</td>
<td>KIFImplication with terminal as symbol</td>
</tr>
<tr>
<td>RULEGENERAL</td>
<td>KIFImplication with arbitrary symbol</td>
</tr>
</tbody>
</table>

Table 5.1: Additional GDL AST Elements
5.2 Knowledge Base

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRoleNames</td>
<td>role names within a string array</td>
</tr>
<tr>
<td>getInitArguments</td>
<td>KIFTerm objects associated with the available init statements</td>
</tr>
<tr>
<td>getRoleObjConst</td>
<td>role names as KIFObjConst objects</td>
</tr>
<tr>
<td>getRaw ...</td>
<td>specified ASTNode type in a KIFSeq list</td>
</tr>
</tbody>
</table>

Table 5.2: GDLForm Helper Methods

Usually the result of calling a parse method on the AST would lead to a KIFForm object. However since the factory methods are changed the result of such a call on a GDLAST leads to an GDLForm object. Since both classes are only returning ASTNode objects, a class cast is needed upon retrieving the root ASTNode. The KIFForm counterpart GDLForm extends its base class with getter methods to retrieve special subsets of all sentences as listed in Table 5.2. The getRaw methods return the specified ASTNode type in a KIFSeq list. For example getRawInits returns all KIFRelSent objects that have “init” as symbol, converts them into Init objects and returns them in a list. This is primarily meant to ease the querying and analysis of the syntax tree on the game description level. More methods are shown in Table 5.2.

5.2 Knowledge Base

Since we are now one level above the raw KIF language we can query the syntax tree in more detail to add more specific information about the available predicates. To enable this, the GDL plug-in adds filter and analyser to the knowledge base part of the KIF plug-in. Whereas the StepCounterFilter is merely for convenience, the classes AUnaryFunctions and AEnvironment add additional properties to the knowledge base. The knowledge base itself contains the static sets of GDL keyword strings (gdlKeywords) and operators (gdlOperators). Both sets are case insensitive tree maps that can be used to check if string constants are contained in one of the sets.

A simple example how the additional information can be used, is given in form of the AEnvironment analyzer. Currently it only checks whether all implications in which a specific predicate is occurring in the head are only depending on either the predicate itself, true statements or on predicates that were already found to be environment look-ups. The term environment look-up refers to the situation

25
that only the knowledge about a game state but no knowledge about moves of other players is necessary to resolve the predicate in question. This knowledge could be used to apply a different mechanism instead of the resolution method which most resolvers are choosing. It is also possible to distinguish different parts of the game, for example tokens that get flipped only based on a specific game situation. Knowing about it might lead to algorithms that remove that token from the game state which they are currently resolving, but keep track of it in a different place. If a predicate has been found to be an environment look-up it is marked by adding the property “isEnvironmentLookup” to its corresponding KBEntry.

Definition 5.1. **Toggle and Counter**

**Toggle**
A toggle is a unary function. It is only occurring in next rules (implications containing a next predicate) and all of these rules are ground.

**Counter**
A counter is a unary function which only has one rule.
It operates on a domain with arity 2. The body of the counter rule consists of one true predicate with the counter function and one predicate which works on a strict ordered domain.

**Step Counter**
The step counter is a counter that ends the game if it reaches a certain state.

As the name **AUnaryFunction** suggests this analyzer is only looking at unary functions and properties of such. As already discussed, finding game elements like the step counter is helping in resolving new game states and finding best next moves. This analyzer is able to find certain types of toggles and counters. Definition 5.1 shows how Palamedes is identifying them. If it finds a toggle it adds the property “isToggle” and the property “domain_distinct” to the knowledge base entry. The “domain_distinct” would not have been found by the **ADomainInformation** class since this one only looks for fact domains. However the nature of the properties content is the same. In case the analyzer finds a counter it adds “isCounter” to the knowledge base entry as well as the property “successor” which identifies the successor predicate by providing the corresponding symbol. Like in the case of a toggle, the analyzer saves the successor’s domain in the property “domain_distinct”. In both cases this property contains the distinct set of all occurring object constants. If available the class denotes the start and end constant in “counterStart” and “counterEnd”. To find the direction a counter counts, the analyzer looks for the argument positions. The first argument of the underlying successor relation is assumed to be left of the second argument within the ordered domain and therefore the first possible constant in argument one would be the start and the other would be the end of the sequence that is defined by the ordered structure.
The implementation furthermore adds some special functionality for these game components. A toggle can show cyclic or acyclic behavior based on the next rules. For counters the cyclic behavior depends on the binary predicate. If its domain does not have a start/end element then the counter will cycle through it. Also it is not necessary that the rules have to fire on each step which is true for the counter as well. One can easily imagine a counter that only is raised if some situation occurs on a game board. For example, there is a rule in Chess that when all player choose to move the same combination of moves more than three times in a row, the match ends in a draw.

If the \texttt{ANARYFUNCTION} analyzer finds a counter, that is set in an init statement and triggers a terminal state, it sets the property \texttt{isStepCounter} in the corresponding knowledge base entry. To ease the finding of this counter predicate, a special filter class is provided, named \texttt{StepCounterFilter}. It basically looks if a predicate contains the property \texttt{isStepCounter} and resolves to true if that is the case.

\section{Resolver}

The resolver is the core of the agent engine. It generates the legal move list and the next states. Since Palamedes is an IDE and but not a GGP agent in general it does not come with an own resolver, but it can make use of existing ones. Currently it supports three different resolvers. With JavaProver and Jocular two java based developmental resolvers are supported. Additionally an adapter for the Fluxplayer was implemented.

However none of them satisfies all needs. Whereas the first two are straightforward to use since they run in the same JVM, incorporating the Fluxplayer adapter takes a little bit more effort to get it started. This is mostly due to Fluxplayers running environment which is Prolog. During testing a couple of issues came up. JavaProver, being the slowest, seems to be the most convenient to use and therefore is the default resolver. Jocular is much faster then the JavaProver but its parser lacks the ability to handle special characters like + even though this is allowed by the KIF specification. This renders some valid source code unusable for simulation since the resolver will parse the code with its own parser.
Chapter 5  GDL Core plug-in

5.4 Game Model

This section will explain how the formal definition of the game model (see Definition 4.1) translates into Palamedes GDL components.

In the first step all games are created through the `GameFactory`. The factory can either load a game description from a file (`loadGame`) or create the game directly from a given string (`createGame`). In both cases it returns an object that implements the `IGame` interface. The concrete implementation of this object is depending on the resolver adapter implementation which can be found in `gdl.core.resolver`.

The resulting game object implements `IGame` which follows the facade design pattern. It hides the game information storage, the game analysis, the reasoning machinery and the game state space representation. All these aspects are accessible through objects that are provided by classes that implement the game model interfaces to keep the internal representations decoupled (Table 5.3). This should help to improve certain aspects of the game mechanics without touching unrelated implementations. Thus the game object is the central structure for all game playing related activities and components.

The set of roles can be directly accessed through the `getRoleNames` method which returns the array of role name strings. To get the set of states one needs to get the game tree first by calling `getTree`. The method returns a game tree object. It contains a tree structured set of game nodes which contain the actual game state. It is of particular importance to understand the distinction between nodes and states. The nodes are components of the tree structure which carry information e.g. about how long is the path between the current node and the root node. It is therefore a

<table>
<thead>
<tr>
<th>Interface</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IGAME</code></td>
<td>Facade for all game functionality</td>
</tr>
<tr>
<td><code>IReasoner</code></td>
<td>Interface for reasoner adapter implementations</td>
</tr>
<tr>
<td><code>IFluent</code></td>
<td>Fluent adapter interface, provided by a reasoner</td>
</tr>
<tr>
<td><code>IMove</code></td>
<td>Move adapter interface, provided by a reasoner</td>
</tr>
<tr>
<td><code>IGameTree</code></td>
<td>Interface for game tree implementations</td>
</tr>
<tr>
<td><code>IGameNode</code></td>
<td>Interface for a minimal game node functionality</td>
</tr>
<tr>
<td><code>IGameState</code></td>
<td>Interface for a minimal game state functionality</td>
</tr>
<tr>
<td><code>IStatistic</code></td>
<td>General Interface for statistic support</td>
</tr>
</tbody>
</table>

Table 5.3: Game Model Interfaces
5.4 Game Model

<table>
<thead>
<tr>
<th>Game Component</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>set of roles</td>
<td>IGame.getRoleNames</td>
</tr>
<tr>
<td>set of states</td>
<td>IGameTree (implicit)</td>
</tr>
<tr>
<td>set of actions</td>
<td>IGame.getLegalMoves and</td>
</tr>
<tr>
<td></td>
<td>IGame.getCombinedMoves</td>
</tr>
<tr>
<td>initial state</td>
<td>IGameTree.getRootNode</td>
</tr>
<tr>
<td>legality relation</td>
<td>IReasoner.getLegalMoves</td>
</tr>
<tr>
<td>update function</td>
<td>IReasoner.getNextState</td>
</tr>
<tr>
<td>terminal states</td>
<td>IGame.isTerminal</td>
</tr>
<tr>
<td>goal values</td>
<td>IGame.getGoalValues</td>
</tr>
</tbody>
</table>

Table 5.4: Access to Game Components

unique instance that can not exist twice in the same tree. The game state however is the description about a particular state in which the game is. It corresponds to the state given by the formal definition. A game state does not know anything about a previous state and of course could occur multiple times in different branches of the game state tree structure. The initial state of the game tree gets usually resolved while creating the game object itself and is available by calling getRootNode of the game tree object.

With the help of the game nodes the game object can be queried to retrieve the legal moves of a certain game state. In the background and if the legal moves are not already known, the reasoner creates them on the fly by resolving them role by role. Then the found legal moves are saved within the game state. To retrieve them, the getLegalMoves method can be used. This method returns an array of combined moves. Combined move means that a cross product of all role dependent moves is created. Therefore each combined move is composed of one move per role thus representing the set of actions. Calling this method might result in a too large move table if a game has a high branching factor and is not a single player game. For this reason the alternative method getCombinedMoves is provided that returns all moves in a special list object. This special list does not contain the fully calculated cross product. Instead it makes an index conversion between the linear index of a normal list and the combined move parts. The resulting combined move array gets created on-the-fly each time it is requested, whereas in the previous method the combined move array only gets calculated once.

Access to the legality relation and the update function is indirectly available through the reasoner adapter by using getLegalMoves and getNextState. The default implementation of Palamedes takes care of utilizing both of them in the background.
so that the strategy implementations do not have to call them directly. Instead the strategy implementations should use methods provided by the \texttt{IGame} interface which in return transparently try to cache as much information as possible to provide a good performance.

Due to the nature of the task the exhaustive set of terminal states and corresponding goal values might not be reasonably computable. This leaves us to provide test methods to check whether a certain state is a terminal one (\texttt{isTerminal}) and if that is the case what the goal value it would have for each player (\texttt{getGoalValues}). Both methods are contained in the game object. An overview about the methods for each mentioned game component can be found in Table 5.4.

5.5 Simulation

So far we are able to create a model of our game at hand and we can go through the states of it by using the resolver. This provides the tools needed to start analysing the game and to create game playing agents. The simulator extends this even further. The simulation part of the plug-in makes it possible to play complete games from start to the end without the need of an external game master. Different strategies can be tested against each other and the simulator can be utilized in the agent implementation to play test games before making a decision about the next move in the current game.

The simulation part is located in \texttt{gdl.core.simulation}. It is a simple game master implementation that skips the HTTP communication part as can be seen in Figure 5.1. Upon creation the simulator receives the GDL description of the game and
5.5 Simulation

<table>
<thead>
<tr>
<th>Class</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractStrategy</td>
<td>abstract skeleton IStrategy implementation,</td>
</tr>
<tr>
<td></td>
<td>inherited by all implementations</td>
</tr>
<tr>
<td>SRandom</td>
<td>random move selection</td>
</tr>
<tr>
<td>SDFS</td>
<td>depth-first search</td>
</tr>
<tr>
<td>SMonteCarlo</td>
<td>monte carlo search</td>
</tr>
<tr>
<td>SMonteCarloUTC</td>
<td>monte carlo search with UTC</td>
</tr>
</tbody>
</table>

Table 5.5: Example Strategy Implementations

distributes it to the match instances (see below). If additional settings are given by a SimulationSetting object the matches will be created accordingly otherwise a random setup will be used.

The Match can be seen as the player agent. It contains the game model, the strategy that the agent is utilizing, the name of the role that this agent should play for and several calculation timers that tells the agent to stop thinking before it hits a deadline. Additionally the match keeps track of all the moves done. The reason to call it “match” instead of “player agent” is, that after the game terminates the match is finished and can not be used to play another game.

The Match gets to know the IStrategy interface only and retrieves the concrete implementation through the StrategyFactory. That makes it easy to implement different strategies that can be interchanged. The strategy receives a specific match and through it can access the underlying game model. Palamedes IDE comes with a couple of strategies which are already implemented and should serve as example implementations (Table 5.5). They can be found in gdl.core.simulation.strategies. How to implement your own strategy is explained in chapter 7.

After the setup the simulator can be started and runs a single simulation. This can be done stepwise or till a terminal state is reached. Basically it iterates over the match objects, lets them generate their internal state and gathers the resulting next moves. The next moves are combined and fed back to the matches and then the cycle starts again till a terminal state is reached. While going through the game, the simulator keeps track of the moves done by the individual matches and stores them into StepInfo together with data for statistic information generation. After all relevant information is retrieved from the simulation object it can be reset to run another simulation with the same settings.
6 UI plug-in

The UI plug-in is the high level part of Palamedes, which provides the interface for user interaction. This is the only real Eclipse plug-in, meaning that this part is not usable outside of Eclipse. If one views the other plug-ins as backend the UI plug-in is the frontend and therefore does not add any additional functionality to the game mechanisms.

6.1 Editors

Palamedes comes with four different editors. Currently, except for the first one, the term “editor” is not quite fitting since only the source code editor can be used to edit the game description. Nonetheless since the other three components in this section can be extended and at the beginning of the project were meant to be real editors, they are grouped into this section as well.

6.1.1 Source Code

The central user interface component of the Palamedes IDE is the source code editor. Even though it currently has not much functionality of its own it is the component, which manages the valid abstract syntax tree and represents the model, all other components work on.

After loading a *.kif or *.gdl file it shows the syntax highlighted source code. The highlighting distinguishes three different colour classes. Variables marked blue, GDL keywords {init, role, next, does, legal, true} marked green and the logical operators {not, and, or, distinct, \(\leq\)} are highlighted red.

Each change in the source code triggers a reconstruction of the abstract syntax tree in the background. While not all changes lead to a valid parse, e.g. in the middle of writing a new rule, the last valid tree is kept and used for all other components until the source code becomes valid again.
6.1 Editors

Figure 6.1: Source Code Editor with Outline & Knowledge Base View

6.1.2 Graph View

This editor shows the rule graph according to [6]. Selecting one or more elements in the outline view will trigger the generation of the corresponding graph. This editor is a proof of concept to show the principles of the rule graph in-situ. Currently an automatic layout generator is used and therefore the resulting graphs, especially large ones, lack a clear arrangement. Since an appropriate automatic layout algorithm will be quite difficult to find this will be complemented with a semi-automatic approach in which the user can correct the layout after getting a suggestion. This part is left for future work since it will involve a lot of effort on this component alone.

6.1.3 Game Tree

While writing the game description, one might wonder whether the game works according to the expectations. To provide a first and easy to use debug tool the game tree editor was created, which shows the game states in a tree view. This
tree then can be expanded by hand. In the background the standard resolver will generate legal moves and new game states on demand. Since game states can have a lot of fluents only positive and negative effects are shown. In this context, positive and negative effects are fluents that got added and removed respectively from the previous game state.

6.1.4 Simulation

The simulation editor is a user interface for steering a simulation. Like the other components it uses the source code as working model. It feeds the abstract syntax tree into the game modeling machinery and creates a game object. With the information of the game, the simulation editor presents the user a settings pane in which for each role a separate strategy can be selected together with the start and play clock as shown in Figure 6.3. A value of -1 for the clocks means that the strategies would have unlimited time to do their work. But be aware that this can lead to a locked interface if the strategy for example does an exhaustive search.

The simulation can run either in a step-by-step mode using the Step button or the
game can be simulated at once by using the Start button. After the start of a whole simulation a Stop button will appear to stop the running simulation. If a simulation gets stopped this way it can be further analysed with the step-by-step mode. However, using the Start button again will start a fresh simulation run. For each game step, a row with a step counter, the positive and negative effects and the played moves is presented in the table on the bottom. Selecting a single row will show the game state in the game state view, if one is open. Also the corresponding resolver statistics of the state is shown in the statistic view.
Chapter 6 UI plug-in

6.2 Views

The Palamedes editors are complemented by several views that can be used in conjunction with the editors.

6.2.1 Outline

The outline view shows the outline of the source code, which is currently open in the editor, as shown in Figure 6.1 on the right side. The first level orders the predicates by name. After opening, it shows the corresponding source code elements. Selecting such an item focuses the editor on this item and selects it in the source code. Source code changes are reflected as soon as a new valid parse of the document is available.

6.2.2 Knowledge Base

Probably being the most complex view, this one shows a lot of information about the game description as one can see in Figure 6.1. It summarised the data contained in the knowledge base and can be expanded by additional analyser implementations. On its first level it distinguishes between function and relation symbols. The second level lists the symbols with their arity. In the left column of the symbol name indicator icons will appear if the knowledge base analyzer manages to identify certain properties of that symbol. Figure 6.4 gives an overview about the symbols and their meaning. In the right column some general information is given through the use of flags. The most simple flag is the TLN which stands for top level nodes and means that the symbol appears in a rule that does not have an implication body and therefore is true at all times. This is used for the init predicate and for constructs like successor functions. IMP gives the number of symbol occurrences in an implication head respective implication body. EL stands for environment look-up.
and indicates a symbol whose current state is only dependent on the current fluents given by the game state. If some information about the domain of the predicate arguments is known, the \( D \) flag will appear. It indicates that the analyzer managed to find out the ranges of the argument domains. Currently that is only done for directly given domains and it is not inferred through a dependency graph or similar methods. An implementation solving that issue could be an interesting extension. The third and last level shows all the known data about the symbol in a “key: value” manner.

To get a better understanding of the constructs that are modeled by the game description, a couple of filters are incorporated into the knowledge base view. The filters are working on the flags mentioned above. It is possible to remove symbols that have top-level nodes, occur in implication heads or bodys. Also GDL keyword symbols can be removed from the view. If it turns out that other flags might be useful for filtering as well, then this might be incorporated in following versions of the IDE.

6.2.3 Game State

The game state view shows the content of a single game. In its basic version it shows a list of the fluents with their arguments ordered alphabetically, as can be seen in Figure 6.2. Each row denotes a single fluent, however each fluent name only appears once to get a clearer overview. If the corresponding game is of known type a stylesheet can be applied to the game state view. The stylesheet then renders a game dependent graphical visualization of the game state. Figure 6.3 shows an example of the visualization of a chinese Checkers board. The same can easily be done for other board games. Especially Chess board based games are easy to adapt.

The game state view works on all kinds of game state sources. It can display states from the game tree editor as well as the states from the simulator editor. If a game state is selectable then it can be displayed in the game state view. The stylesheet looks for known fluent names and tries to match them. Therefore this approach only works if the game is not scrambled. Games of the wrong type but with “correct” fluent names would simply result in a bugged visualization. The stylesheets can be managed in the preferences section of Palamedes IDE. They can be edited and imported from an external source.
6.2.4 Statistics

As stated before the simulation is able to gather statistic data while running. The statistic view can be used in conjunction to the simulation editor to show the internally gathered data. After selecting a single step row in the simulator the view shows the corresponding data snapshot (see Figure 6.3).

The following information will be displayed:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legal Move Count</td>
<td>The number of legal moves that got generated.</td>
</tr>
<tr>
<td>Legal Move Calls</td>
<td>The number of calls of \texttt{IReasoner.getLegalMoves}</td>
</tr>
<tr>
<td>Legal Move Time (ms)</td>
<td>The time spend for legal move calculation in ms</td>
</tr>
<tr>
<td>Next Node Calls</td>
<td>The number of calls of \texttt{IGame.getNextNode}</td>
</tr>
<tr>
<td>Next State Calls</td>
<td>The number of calls of \texttt{IReasoner.getNextState}</td>
</tr>
<tr>
<td>Next State Time (ms)</td>
<td>The time spend for next state calculation in ms</td>
</tr>
</tbody>
</table>
7 Extensions

Palamedes was written with the idea in mind of supporting the development process of GGP systems. Therefore it comes with a number of extension points to incorporate programming efforts from the science community. A nice example of this are the adapters for the JavaProver and Jocular from the Stanford University and for FluxPlayer from our lab. One driving force to implement or even reimplement parts of the system is the possibility to compare different approaches and to use Palamedes as a benchmarking framework. So far no other system is known to us that can offer this kind of service. More often than necessary a lack of comparability leads to short-lived solutions. We will show that this can be easily overcome by using Palamedes.

Extending the framework is done by implementing interfaces and thus providing adapter classes for the specific functionality. These classes are registered at a factory which then creates the corresponding objects. This way only the functionality in question has to be implemented without bothering how it is used by the system.

7.1 Resolver Adaptation

To incorporate an own resolver the following interfaces have to be implemented:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Abstract Base Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGame</td>
<td>GAME</td>
</tr>
<tr>
<td>IGameState</td>
<td>ABSTRACTSTATE</td>
</tr>
<tr>
<td>IMove</td>
<td>GENERICMOVE</td>
</tr>
<tr>
<td>IFluent</td>
<td>GENERICFLUENT</td>
</tr>
<tr>
<td>IReasoner</td>
<td>none</td>
</tr>
</tbody>
</table>

While using the default game model the abstract base classes can be used to derive the necessary adapter classes.

We start with the TERMADAPTER and FLUENTADAPTER classes since they are the most basic ones in this example. The TERMADAPTER class needs to take care of the translation between the reasoner syntax tree elements and the KIF syntax tree elements. An example is shown in Listing B.2 (line 12-25). The FLUENTADAPTER then needs
to use the term adapter to translate fluents. This can be done in the construction of the fluent as shown in Listing B.1.

In the next step the **StateAdapter** can be implemented. In our example (Listing B.3) it inherits the **AbstractState** class, which in return is a basic implementation of the **IGameState** interface. The **StateAdapter** class is the game state information carrying class on the Java Prover side. This one contains the game state fluents in a map object that can be used to iterate over all fluents. The only thing that needs to be done here is to decorate a **Predicate** with a **FluentAdapter** such that the predicate behaves like a fluent in our system. The most complicated part is the implementation of the **Reasoner** interface, which is highly reasoner dependent and therefore here only shown as an example implementation (Listing B.5).

### 7.2 Game Model Interface

If replacing the available resolver is not sufficient and in the case that the provided game tree implementation needs to be changed one can even replace the game engine by implementing the game model interfaces given in Table 7.1.

Currently, the Palamedes implementations of the interfaces **IGame (Game)** and **IGameState (AbstractState)** are utilised to support the reasoner implementation (see above). Thus, changing one of these classes could break the reasoner adapter implementations. Nevertheless, the core of the game engine, the game tree, can be replaced with your own version.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Palamedes Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGame</td>
<td>Game</td>
</tr>
<tr>
<td>IGameTree</td>
<td>GameNodeTree</td>
</tr>
<tr>
<td>IGameNode</td>
<td>GameNode</td>
</tr>
<tr>
<td>IGameState</td>
<td>AbstractState</td>
</tr>
</tbody>
</table>

Table 7.1: Game Model Interface Implementations

The **IGAMETREE** is kept simple to ease the adaptation. The interface only requires the developer to provide methods to retrieve the root node of the tree (**getRootNode**), to add child nodes (**addChild**), to traverse the tree (**getNextNode**) and to mark a single node to contain a terminal state (**setTerminal**). This interface only allows for implementations that traverse the tree in a downwards manner, from a parent to the child nodes. Therefore the interface of the actual game node **IGAMENODE** requires an
implementation for retrieving the parent node (getParent) and the depth of the node within the game tree (getDepth). Besides these two, only methods for the internal storage of the game state are necessary. An example of a game tree implementation is given by the GameNodeTree and GameNode classes in the gdl.core.utils package of Palamedes.

### 7.3 Strategy Interface

Probably being the most important extension point the strategy interface is kept as small as possible as one can see in Figure 7.1. It is meant to be easy to provide own implementations of strategies. Some examples are already given in the gdl.core.simulation.strategies package.

With the AbstractStrategy class Palamedes provides an abstract class that already covers some of the most common properties necessary for strategy implementations. For a quick start it is sufficient to provide only the initMatch and getMove methods of the interface while inheriting from AbstractStrategy.

As last call during the construction of a match object the method initMatch of a strategy gets called. Therefore the passed match object can be used to its full extent to derive the game object and with it all game relevant information. The implementation of this method should contain all setup code for the strategy implementation.
and can be used for the match start time calculation. The play time calculations are done within the `getMove` method that needs to return a selected move. This function should under no circumstances return anything other then a valid move. If a strategy implementation is able to calculate an expected goal value for its move selection then this value can be delivered by the `getExpectedGoalValue`.

Heuristic strategies can implement the `getHeuristicValue` so that player implementations can access this information in case more then one strategy is used to determine the next moves. These heuristic values should only reflect the internal values that an individual strategy is applying and can only be used to compare different implementations of the same strategy type. To compare the quality of the move selection across different strategies the method `getReliabilityValue` should be used in the future. However, currently this method is only a stub since a good algorithm to calculate such reliability values is not available yet.

Once a new strategy implementation is written it needs to be registered at the `StrategyFactory`. To do so one needs to create a `StrategyDescription`. The constructor of this class gets the name, the full class name of the strategy and a short description string that gets displayed in the user interface to give some indication what the strategy is about to do. After creating the description object it only needs to be fed to the factory by calling its `addStrategy` method (Listing 7.1).

```
1  StrategyFactory.getInstance().addStrategy(
2       new StrategyDescription(  "Random",
3       StrategyFactory.defaultPackage + ".SRandom",
4       "Plain Random Move Choice" )
    );
```
Listing 7.1: Register a strategy implementation.
8 Conclusions

With the Palamedes IDE we have developed an Integrated Development Environment and research platform for the GGP community. Since it is the first of its kind there is still a lot of work that needs to be done until Palamedes can be considered full-featured and fully functional.

We have shown that Palamedes is able to support the writing of game descriptions by supplying an outline tree and the syntax highlighting which both are generated on-the-fly and thus make it easier to spot errors in the descriptions. Also getting an overview of the content of a description file is much easier with such tools.

To aid the creation of new game descriptions even further, a template system could be implemented that provides skeleton descriptions including basic game components like a step counter or boards of various sizes and dimensions. Together with the templates, a source code refactoring tool could be provided which helps with renaming of symbols or insertion of game components. Both approaches would help to reduce errors in handwritten source code due to semi-automatic code generation. Nonetheless errors still can be introduced into the code by the developer. For improvement, a quick help mechanism and a more detailed parser, reporting source code problems in a better way should be developed.

For the task of understanding the content of game descriptions wrt. the intrinsic game features, we have provided the knowledge base. It is already able to reason about certain aspects like finding the step counters but foremost it is extendable. It is possible to incorporate approaches and properties not available so far. Also the teaching aspect should be underlined here since everyone can supply an own analysis method with the tools at hand without writing all the necessary steps in between like the parser or an editor. With the graph view, a first step towards visualising game components has been made. We believe that visualise games might lead to feature discoveries that can not intuitively seen by looking at the written rules.

Additionally, we have provided a framework for developing player agents together with a simulation environment as a test-bed for strategy implementations. The provided examples show how the whole framework interacts and which parts can be replaced by own, more suitable implementations. Especially the implementation of different additional strategies can aid the teaching aspect since with Palamedes a platform is available which can be used to compare strategies by only replacing them
while keeping the rest of the agent the same way. Also comparing different reasoner
and their optimizations is easier if they get incorporated into Palamedes.

Last but not least it needs to be stated that providing the Palamedes IDE as Eclipse
plug-ins seemed to be convenient at first but has shown the drawback that it is
not recognisable as tool by itself. For this reason and to avoid the installation
procedures we plan to turn it into an Eclipse Rich Client Program in one of the next
versions.

However at this point it already is a very useful tool that can be utilized for devel-
oping new game descriptions, to analyse existing ones, for writing own players and
for analyzing different strategy and reasoner implementations.
A Parser Definition

Since no useable KIF parser definition was available we include here our own one to offer a starting point for own implementations. Both descriptions need to be fed to the CUP parser generator which results in three generated classes (KIFSsymbols, KIFScanner and KIFParser). They can be found in the kif.core.parser package.

```
package org.eclipse.palamedes.kif.core.parser;
import java_cup.runtime.*;

%%
% class KIFScanner
%cup
%ignorecase
%line
%column
%full
%eofval{
  return symbol(KIFSsymbols.EOF);
%eofval}
%
{ private final Symbol symbol(int type) {
      return new Symbol(type, zzStartRead, zzMarkedPos);
    }
  private final Symbol symbol(int type, Object value) {
      return new Symbol(type, zzStartRead, zzMarkedPos, value);
    }
}%

LineTerminator = \r|\n|\r
InputCharacter = [^\r\n] |
WhiteSpace = { LineTerminator }+ | [ \t\f\ ]+
SingleComment = ;" { InputCharacter }* { LineTerminator }
Comment = { SingleComment }+
Variable = "?" [\_a-zA-Z] [^\t\f\ ]* |
Constant = [^\t\f\ ]*|

%% /* keywords */
<YYINITIAL> {
  "(" { return symbol(KIFSsymbols.OPEN); }
  ")" { return symbol(KIFSsymbols.CLOSE); }
  "not" { return symbol(KIFSsymbols.NOT); }
  "and" { return symbol(KIFSsymbols.AND); }
  "or" { return symbol(KIFSsymbols.OR); }
  ">=" { return symbol(KIFSsymbols.IMP); }
  {Variable} { return symbol(KIFSsymbols.VARIABLE, yytext()); }
  {Constant} { return symbol(KIFSsymbols.CONSTANT, yytext()); }
  {Comment} { return symbol(KIFSsymbols.COMMENT, yytext()); }
  {WhiteSpace} {}
}
```
Listing A.1: Scanner Definition (KIF.flex)

```java
import java_cup.runtime.Symbol;
import org.eclipse.palamedes.kif.core.ast.*;

parser code {
    AST ast = new AST();

    /* Constructor that replaces the default AST with an own one. */
    public KIFParser(AST ast) {
        super();
        this.ast = ast;
    }

    public void report_error(String message, Object info) {
        System.err.print(message);
        if (info instanceof Symbol)
            if (((Symbol)info).left != -1)
                System.err.println(" at character " + ((Symbol)info).left + " of input with value " + ((Symbol)info).value + " expected symbol type " + ((Symbol)info).sym);
            else System.err.println("");
        else System.err.println("");
    }
}

terminal OPEN, CLOSE, NOT, AND, OR, IMP;
terminal String VARIABLE, CONSTANT, COMMENT;
non terminal String objconst, logconst, relconst, funconst;
non terminal KIFForm form;
non terminal KIFSent sentence;
non terminal KIFRelSent relsent;
non terminal KIFLogSent logsent;
non terminal KIFTerm term;
non terminal KIFFunTerm functerm;
non terminal KIFSeq sentenceseq, termseq;

form ::= sentenceseq: sentenceseq
    { : RESULT = parser.ast.newForm(sentenceseq); : }
    |
    { : RESULT = parser.ast.newForm(); : }
    ;

sentence ::= logconst:logconst
    { : RESULT = parser.ast.newLogConst(logconst); : }
    |
    relsent: relsent
    { : RESULT = relsent; : }
    |
    logsent: logsent
    { : RESULT = logsent; : }
    |
    COMMENT:c
    { : RESULT = parser.ast.newComment(c); : }
    ;

relsent ::= OPEN:o relconst:relconst termseq:termseq CLOSE:c
```
logsent ::= OPEN:o NOT sentence:sentence CLOSE:c
{ RESULT = parser.ast.newNegation(sentence); }  
| OPEN:o AND sentence:sentence sentenceseq:sentenceseq CLOSE:c
{ RESULT = parser.ast.newConjunction(sentence, sentenceseq); }  
| OPEN:o OR sentence:sentence sentenceseq:sentenceseq CLOSE:c
{ RESULT = parser.ast.newDisjunction(sentence, sentenceseq); }  
| OPEN:o IMP sentence:sentence sentenceseq:sentenceseq CLOSE:c
{ RESULT = parser.ast.newImplication(sentence, sentenceseq); }  
| OPEN:o IMP sentence:sentence CLOSE:c
{ RESULT = parser.ast.newImplication(sentence); }  

term ::= VARIABLE:indvar
{ RESULT = parser.ast.newIndVar(indvar);  
RESULT.setSourceStart(indvarleft);  
RESULT.setSourceLength(indvarright - indvarleft); }  
| objconst:objconst
{ RESULT = parser.ast.newObjConst(objconst);  
RESULT.setSourceStart(objconstleft);  
RESULT.setSourceLength(objconstright - objconstleft); }  
| functerm:functerm
{ RESULT = functerm; }  

functerm ::= OPEN:o funconst:funconst termseq:termseq CLOSE:c
{ RESULT = parser.ast.newFunTerm(funconst, termseq);  
RESULT.setSourceStart(oleft);  
RESULT.setSourceLength(cright - oleft); }  
| OPEN:o funconst:funconst CLOSE:c
{ RESULT = parser.ast.newFunTerm(funconst);  
RESULT.setSourceStart(oleft);  
RESULT.setSourceLength(cright - oleft); }  

objconst ::= CONSTANT:objconst
{ RESULT = objconst; }  
| funconst ::= CONSTANT:funconst
{ RESULT = funconst; }  
| relconst ::= CONSTANT:relconst
{ RESULT = relconst; }  
| logconst ::= CONSTANT:logconst
{ RESULT = logconst; }  

sentenceseq ::= sentenceseq:sentenceseq sentence:sentence
{ RESULT = parser.ast.newSeq(sentenceseq, sentence); }  
| sentence:sentence
{ RESULT = parser.ast.newSeq(sentence); }  

termseq ::= termseq:termseq term:term
{ RESULT = parser.ast.newSeq(termseq, term); }  
| term:term
{ RESULT = parser.ast.newSeq(term); }  

Listing A.2: Parser Definition (KIF.cup)
B Example Resolver Adaptation with JavaProver

Providing an adapter for your own resolver is quite easy. Here it is shown by the example of the Javaprover adapter. Detailed instructions are given in chapter 7. Import and package statements are not shown here.

Listing B.1: gdl.resolver.javaprover.FluentAdapter.java

```java
public final class FluentAdapter extends GenericFluent<Expression> {
    public FluentAdapter(Expression exp) {
        super(new TermAdapter(((Connective)exp).getOperands().get(0));
    }
}
```

Listing B.2: gdl.resolver.javaprover.TermAdapter.java

```java
public class TermAdapter extends TermWrapper<Expression>{
    public TermAdapter(Expression nativeTerm) {
        super(nativeTerm);
    }

    public KIFTerm getKIFTerm() {
        return getKIFTerm(nativeTerm);
    }

    public static final KIFTerm getKIFTerm(Expression expr) {
        if (expr instanceof Variable) {
            return new AST().newIndVar("v" + Long.toString(((Variable)expr).getNum()));
        } else if (expr instanceof Atom) {
            return new AST().newObjConst(expr.toString().toLowerCase());
        } else if (expr instanceof Connective) {
            Connective c = (Connective)expr;
            KIFSeq<KIFTerm> args = new KIFSeq<KIFTerm>();
            for (Expression e:(List<Expression>)c.getOperands().toArrayList()) {
                args.add(getKIFTerm(e));
            }
            return new AST().newFunTerm(c.getOperator().toString().toLowerCase(),
                   args);
        }
        return null;
    }
}
```

Listing B.3: gdl.resolver.javaprover.StateAdapter.java

```java
public class StateAdapter extends AbstractState<GameState> {
    public StateAdapter(GameState objState) {
        super(objState);
    }

    public void initFluentList() {
    }
}
```
for (Object fluentList : nativeState.getMap().values())
for (Object exp : ((ExpList) fluentList).toArrayList())
  fluents.add(new FluentAdapter((Expression) exp));
}
}

Listing B.3: gdl.resolver.javaprover.StateAdapter.java

class GameJP extends Game {
  public GameJP(String gdl) {
    super(new ReasonerAdapter(new GameSimulator(false, false)),
          new GameNodeTree());
    sourceGDL = gdl;
    // Initialize simulator
    ReasonerAdapter rAdapter = ((ReasonerAdapter) reasoner);
    rAdapter.simulator.ParseDescIntoTheory(sourceGDL);
    // get role names
    ExpList roles = rAdapter.simulator.GetRoles();
    roleName = new String[roles.size()];
    int i = 0;
    for (Object role : roles.toArrayList())
      roleName[i++] = role.toString().toLowerCase();
    // set init node
    rAdapter.simulator.SimulateStart();
    StateAdapter state = new StateAdapter(rAdapter.simulator.GetGameState());
    if (rAdapter.simulator.IsTerminal()) {
      state.setTerminal();
    }
    ((GameNodeTree) tree).setRoot(state);
  }
}

Listing B.4: gdl.resolver.javaprover.GameJP.java

class ReasonerAdapter implements IReasoner {
  GameSimulator simulator;
  private static final class MoveAdapter extends GenericMove<Expression> {
    public MoveAdapter(Expression exp) {
      super(new TermAdapter(exp));
    }
  }
  public ReasonerAdapter(GameSimulator sim) {
    simulator = sim;
  }
  public int getGoalValue(String roleName, IGameState currentState)
  throws InterruptedException {
    simulator.SetGameState((GameState) currentState.getNativeState());
    Theory theory = simulator.getTheory();
    ExpList goalArgs = new ExpList();
    goalArgs.add(new Atom(roleName));
    Variable vX = new Variable("X");
    goalArgs.add(vX);
    Expression e;
  }
try {
    e = theory.findIndex(vX, new Predicate(new Atom("goal"), goalArgs));
} catch (InterruptedException ie) {
    e = null;
}
if (e == null) return -1;
else return Integer.parseInt(e.toString());

public IMove[] getLegalMoves(String roleName, IGameState currentState)
throws InterruptedException {
    simulator.SetGameState((GameState) currentState.getNativeState());
    ExpList legalMoves = simulator.GetLegalMoves(new Atom(roleName));
    if (legalMoves == null) {
        System.out.println("Warning: null result from Reasoner!");
        return new IMove[0];
    }
    IMove[] moves = new IMove[legalMoves.size()];
    for (int i = 0; i < legalMoves.size(); i++)
        moves[i] = new MoveAdapter((Predicate) legalMoves.get(i));
    return moves;
}

public IGameState getNextState(IGameState currentState, IMove[] moves)
throws InterruptedException {
    // create native move list
    ExpList moveList = new ExpList();
    for (IMove move : moves)
        moveList.add(((MoveAdapter)move).getNativeMove());
    // generate next state
    simulator.SetGameState((GameState) currentState.getNativeState());
    simulator.SimulateStep(moveList);
    StateAdapter state = new StateAdapter(simulator.GetGameState());
    if (simulator.IsTerminal())
        state.setTerminal();
    return state;
}

public IMove createMove(String moveStr) {
    return new MoveAdapter((Predicate)Parser.parseDesc(moveStr).get(0));
}
Bibliography


Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den

__________________________
Ingo Keller