Chapter 4
Programming Rational Agents in GOAL

Koen V. Hindriks

Abstract The agent programming language GOAL is a high-level programming language to program *rational agents* that derive their *choice of action* from their *beliefs* and *goals*. The language provides the basic building blocks to design and implement rational agents by means of a set of programming constructs. These programming constructs allow and facilitate the manipulation of an agent's beliefs and goals and to structure its decision-making. GOAL agents are called rational because they satisfy a number of basic *rationality constraints* and because they decide to perform actions to further their goals based upon a reasoning scheme derived from *practical reasoning*. The programming concepts of belief and goal incorporated into GOAL provide the basis for this form of reasoning and are similar to their common sense counterparts used everyday to explain the actions that we perform. In addition, GOAL provides the means for agents to *focus their attention* on specific goals and to *communicate* at the *knowledge level*. This provides an intuitive basis for writing high-level agent programs. At the same time these concepts and programming constructs have a *well-defined, formal semantics*. The formal semantics provides the basis for defining a *verification framework* for GOAL for verifying and reasoning about GOAL agents which is similar to some of the well-known agent logics introduced in the literature.

4.1 Motivation

The concept of a *goal* lies at the basis of our understanding of why we perform actions. It is common sense to explain the things we do in terms of beliefs and goals. I started writing this chapter with the goal of explaining the programming language GOAL. The *reasons* for performing actions are derived from our moti-

Koen V. Hindriks
EEMCS, Delft University of Technology, Mekelweg 4, Delft
e-mail: k.v.hindriks@tudelft.nl

vations and the notion of *rational behaviour* is typically explained in terms of actions that are produced in order to further our goals [5, 14, 16]. A researcher that has a goal to have finished a book chapter but is going on a holiday instead is not considered to behave rationally because holidays do not further the goal of writing a book chapter.

The idea to use common sense notions to build programs can be traced back to the beginnings of Artificial Intelligence. Shoham, who was one of the first to propose a new programming paradigm that he called *agent-oriented programming*, cites McCarthy about the usefulness of ascribing such notions to machines [29, 39]. One of the first papers on Artificial Intelligence, also written by McCarthy, is called *Programs with Common Sense* [28]. It has been realized that in order to have machines compute with such notions it is imperative to precisely specify their meaning [39]. To this end, various logical accounts have been proposed, mainly using modal logic, to clarify the core common sense meaning of these notions [10, 25, 34]. These accounts have aimed to precisely capture the essence of a conceptual scheme based on common sense that may also be useful and applicable in specifying rational agent programs. The first challenge thus is to provide a well-defined semantics for the notions of belief, goal and action which can also provide a computational interpretation of these notions useful for programming agents.

One of the differences between our approach and earlier attempts to put common sense concepts to good use in Artificial Intelligence is that we take a definite *engineering stance* (contrast [28] and [39]). The concepts are used to introduce a new agent programming language that provides useful programming constructs to develop agent programs. The second challenge is to provide agent programming language that is practical, transparent, and useful. It must be practical in the sense of being easy to use, transparent in the sense of being easy to understand, and useful in the sense of providing a language that can solve real problems.

4.1.1 The GOAL Agent Programming Language

The agent programming language GOAL that we will introduce and discuss meets both of the challenges identified above [3, 22]. The distinguishing feature of the language GOAL is its notion of *declarative goals* and the way agents derive their choice of actions from such goals.[1] The beliefs and goals of a GOAL agent are called its *mental state*. Various constraints are placed on the mental state of an agent, which roughly correspond to constraints on their common sense counterparts. On top of the mental attitudes a GOAL agent also has so-called action rules to guide the action selection mechanism.

[1] GOAL is an acronym for *G*oal-Oriented *A*gent *L*anguage.

The main features of GOAL include:

- *Declarative beliefs*: Agents use a symbolic, logical language to represent the information they have, and their beliefs or knowledge about the environment they act upon in order to achieve their goals. This *knowledge representation language* is not fixed by GOAL but, in principle, may be varied according to the needs of the programmer.
- *Declarative goals*: Agents may have multiple goals that specify *what* the agent wants to achieve at some moment in the near or distant future. Declarative goals specify a state of the environment that the agent wants to establish, they do not specify actions or procedures how to achieve such states.
- *Blind commitment strategy*: Agents commit to their goals and drop goals only when they have been achieved. This commitment strategy, called a *blind* commitment strategy in the literature [34], is the *default* strategy used by GOAL agents. Rational agents thus do not have goals that they believe are already achieved, a constraint which has been built into GOAL agents.
- *Rule-based action selection*: Agents use so-called *action rules* to select actions, given their beliefs and goals. Such rules may *underspecify* the choice of action in the sense that multiple actions may be performed at any time given the action rules of the agent. In that case, a GOAL agent will select an arbitrary action for execution.
- *Policy-based intention modules*: Agents may focus their attention and put all their efforts on achieving a subset of their goals, using a subset of their actions, using only knowledge relevant to achieving those goals. GOAL provides modules to structure action rules and knowledge dedicated to achieving specific goals. Informally, modules can be viewed as policy-based intentions in the sense of [6].
- *Communication at the knowledge level* [31]: Agents may communicate with each other to exchange information, and to coordinate their actions. GOAL agents communicate using the knowledge representation language that is also used to represent their beliefs and goals.

This brief but comprehensive overview of the GOAL language illustrates the range of concepts that are available to program rational agents. GOAL is a high-level and expressive language that facilitates programming agents that derive their choice of action from their beliefs and goals. Arguably, as the reader may convince his or herself by means of the examples provided below, the language is easy to understand, which is achieved by a careful balance between the rich common sense intuitions associated with these concepts and their formal counterparts that have been incorporated into GOAL. Moreover, transparency is achieved since the programming contructs available do not aim at capturing all the subtle nuances of the rich common sense concepts but only their core meaning.

4.2 Language

In Section 4.2.1, the GOAL language is firstly introduced by means of a number of examples that illustrate what a GOAL agent program looks like. A classical and well-known domain called the *Blocks World* has been used for this purpose. We like to think of the Blocks World as the "hello world" example of agent programming (see also [40]). It is both simple and rich enough to demonstrate various of the available programming constructs in GOAL. In Section 4.2.2 the operational semantics of GOAL is introduced as well as a program logic to verify properties of GOAL agents.

4.2.1 Syntactical Aspects

A GOAL agent decides which action to perform next based on its beliefs and goals. In a Blocks World the decision amounts to where to move a block, in a robotics domain it might be where to move to or whether to pick up something with a gripper or not. Such a decision typically depends on the current state of the agent's environment as well as general knowledge about this environment. In the Blocks World an agent needs to know what the current configuration of blocks is and needs to have basic knowledge about such configurations (e.g. when is a block part of a tower) to make a good decision. The former type of knowledge is typically *dynamic* and changes over time, whereas the latter typically is *static* and does not change over time. In line with this distinction, two types of knowledge of an agent are distinguished: conceptual or domain knowledge stored in a *knowledge base* and beliefs about the current state of the environment stored in a *belief base*.
A decision to act will usually also depend on the goals of the agent. In the Blocks World a decision to move a block on top of an existing tower of blocks would be made, for example, if it is a goal of the agent to have the block on top of that tower. In a robotics domain it might be that the robot has a goal to bring a package somewhere and therefore picks it up. Goals of an agent are stored in a *goal base*. The goals of an agent may change over time, for example, when the agent adopts a new goal or drops one of its goals. As a rational agent should not pursue goals that it already believes to be achieved, such goals need to be removed. GOAL provides a built-in mechanism for doing so based on a so-called *blind commitment strategy*. We will discuss this built-in goal update mechanism in more detail below.
Together, the knowledge, beliefs and goals of an agent make up its *mental state*. A GOAL agent inspects and modifies this state at runtime analogously as a Java method operates on the state of an object. Agent programming in GOAL therefore can also be viewed as *programming with mental states*.
To select an action a GOAL agent needs to be able to *inspect* its knowledge, beliefs and goals. An action may or may not be selected if certain things follow from an agent's mental state. For example, if a block is misplaced, that is, the current position of the block does not correspond with the agent's goals, the agent may

decide to move it to the table. A GOAL programmer needs to write special conditions called *mental state conditions* in order to verify whether the appropriate conditions for selecting an action are met. In essence, writing such conditions means specifying a *strategy* for action selection that will be used by the GOAL agent. Such a strategy is coded in GOAL by means of *action rules* which define when an action may or may not be selected. After selecting an action, an agent needs to *perform* the action. Performing an action in GOAL means changing the agent's mental state. An action to move a block, for example, will change the agent's beliefs about the current position of the block. The effects of an action on the mental state of an agent need to be specified explicitly in a GOAL agent program by the programmer except for a few built-in actions. Whether or not a real (or simulated) block will also be moved in an (simulated) environment depends on whether the GOAL agent has been adequately connected to such an environment. Although there are many interesting things to say about this connection (related to e.g. failure of actions and percepts obtained through sensors), in this chapter we will not discuss this in any detail.
We are now ready to define more precisely what a GOAL agent is. A *basic* GOAL *agent program* consists of five sections: (1) a set of domain rules, which is optional, collectively called the *knowledge base* of the agent, (2) a set of beliefs, collectively called the *belief base*, (3) a set of goals, called the *goal base*, (4) a *program section* which consists of a set of action rules, and (5) an *action specification* that consists of a specification of the pre- and post-conditions of the actions available to the agent. To avoid confusion of the program section with the agent program itself, from now on, the agent program will simply be called *agent*. The term agent will be used both to refer to the program text itself as well as to the execution of such a program. It should be clear from the context which of the two senses is intended. An Extended Backus-Naur Form syntax definition (cf. [38]) of a GOAL program is provided in Table 4.1.[2] The syntax specification of GOAL also contains references to modules. Modules are discussed in Section 4.2.1.2.

4.2.1.1 A GOAL Blocks World Agent

In order to explain how a GOAL agent works, we will design an agent that is able to effectively solve Blocks World problems. To this end, we now briefly introduce the Blocks World domain. The Blocks World is a simple environment that consists of a finite number of blocks that are stacked into *towers* on a table of *unlimited* size. It is assumed that each block has a unique label or name $a, b, c, \ldots$. Labelling

[2] Here, boldface is used to indicate *terminal symbols*, i.e. symbols that are part of an actual program. Italic is used to indicate *nonterminal symbols*. [...] is used to indicate that ... is optional, | is used to indicate a choice, and * and $^+$ denote zero or more repetitions or one or more repetitions of a symbol, respectively. The nonterminal *clause* refers to arbitrary Prolog clauses, which is dependent on the Prolog system used. The current implementation of GOAL uses SWI-Prolog [42]. It is only allowed, however, to use a subset of the built-in predicates available in SWI-Prolog; in particular, for example, no meta-predicates can be used.

program	::=	**main** *id* **{**
		[**knowledge** **{** *clause** **}**]
		beliefs **{** *clause** **}**
		goals **{** *poslitconj** **}**
		program **{** (*actionrule* | *module*)$^+$ **}**
		action-spec **{***actionspecification***}**
		}
module	::=	**module** *id* **{**
		context **{** *mentalstatecond* **}**
		[**knowledge** **{** *clause** **}**]
		[**goals** **{** *poslitconj** **}**]
		program **{** (*actionrule* | *module*)$^+$ **}**
		[**action-spec** **{***actionspecification***}**]
		}
clause	::=	any legal Prolog clause **.**
poslitconj	::=	*atom* {, *atom*}* **.**
litconj	::=	[**not**]*atom* {, [**not**]*atom*}*
atom	::=	*predicate*[*parameters*]
parameters	::=	(*id*{ ,*id*}*)
actionrule	::=	**if** *mentalstatecond* **then** *action* **.**
mentalstatecond	::=	*mentalatom* { , mentalatom }* | **not(** *mentalstatecond*)
mentalatom	::=	`true` | **bel** (*litconj*) | **goal** (*litconj*)
actionspec	::=	action **{ pre{***litconj***} post{***litconj***} }**
action	::=	*user-def action* | *built-in action*
user-def action	::=	*id*[parameters]
built-in action	::=	**insert(** *poslitconj*) | **delete(** *poslitconj*) |
		adopt(*poslitconj*) | **drop(** *poslitconj*) |
		send(*id* , *poslitconj*)
id	::=	(a..z | A..Z | _ | $) { (a..z | A..Z | _ | 0..9 | $) }*

Table 4.1 Backus Naur Syntax Definition

blocks is useful because it allows us to identify a block uniquely by its name. This is much simpler than having to identify a block by means of its position with respect to other blocks, for example. Typically, labels of blocks are used to specify the current as well as goal configurations of blocks, a convention that we will also use here. Observe that in that case labels define a unique feature of each block and they cannot be used interchangeably as would have been the case if only the colour of a block would be a relevant feature in any (goal) configuration. In addition, blocks need to obey the following "laws" of the Blocks World: (i) a block is either on top of another block or it is located somewhere on the table; (ii) a block can be directly on top of at most one other block; and, (iii) there is at most one block directly on top of any other block (cf. [11]).[3] Although the Blocks World domain defines a rather simple environment it is sufficiently rich to illustrate various features of GOAL and to demonstrate that GOAL allows to program simple and elegant agent programs to solve such problems.

[3] For other, somewhat more realistic presentations of this domain that consider e.g., limited table size, and varying sizes of blocks, see e.g. [18].

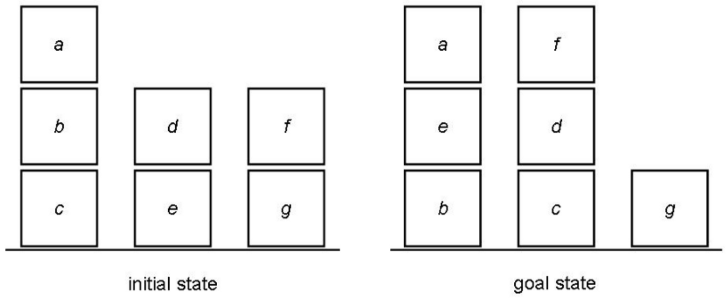


Fig. 4.1 Example Blocks World problem taken from [40].

A *Blocks World problem* is the problem of which actions to perform to transform an initial state or configuration of towers into a goal configuration, where the exact positioning of towers on the table is irrelevant. A Blocks World problem thus defines an action selection problem which is useful to illustrate the action selection mechanism of GOAL. See Figure 4.1 for an example problem. Here we assume that the only action available to the agent is the action of moving one block that is on top of a tower onto the top of another tower or to the table. A block on top of a tower, that is, a block without any block on top of it, is said to be *clear*. As there is always room to move a block onto the table, the table is also said to be clear.
The performance of a Blocks World agent can be measured by means of the number of moves it needs to transform an initial state or configuration into a goal state. An agent performs optimally if it is not possible to improve on the number of moves it uses to reach a goal state.[4] Some basic insights that help solving a Blocks World problem and that are used below in the design of an agent that can solve such problems are briefly introduced next. A block is said to be *in position* if the block in the current state is on top of a block or on the table and this corresponds with the goal state, and all blocks (if any) below it are also in position. A block that is not in position is said to be *misplaced*. In Figure 4.1 all blocks except block c and g are misplaced. Observe that only misplaced blocks have to be moved in order to solve a Blocks World problem. The action of moving a block is called *constructive* if in the resulting state that block is in position. It should be noted that in a Blocks World where the table has unlimited size in order to reach

[4] The problem of finding a minimal number of moves to a goal state is also called the *optimal* Blocks World problem. This problem is NP-hard [18]. It is not within the scope of this chapter to discuss either the complexity or heuristics proposed to obtain near-optimal behaviour in the Blocks World; see [13] for an approach to define such heuristics in GOAL.

the goal state it is only useful to move a block onto another block if the move is constructive, that is, if the move puts the block in position. Also observe that a constructive move always decreases the number of misplaced blocks.[5]

Representing Knowledge, Beliefs and Goals

One of the first steps in developing and writing a GOAL agent is to design and write the knowledge, beliefs and goals that an agent needs to meet its design objectives. The process of doing so need not be finished in one go but may need several iterations during the design of an agent before completing the **knowledge**, **beliefs**, and **goals** sections of a GOAL agent. It is however important to get the representation of the agent's knowledge, beliefs and goals right as both the action specifications and action rules also depend on it. To do so we need a *knowledge representation language* that we can use to describe the content of the various mental attitudes of the agent. Although, as will be explained in Section 4.2.2, GOAL is not married to any particular knowledge representation language, here, *Prolog* will be used to present an example GOAL agent. We assume the reader to be familiar with the basics of Prolog (see [41] for a classic introduction), although familiarity with first-order logic probably will be sufficient to understand the example.
In the Blocks World, first of all we need to be able to represent the configuration of blocks. That means we need to be able to represent which block is on another block and which blocks are clear. In order to do so, the expressions `on(X,Y)` and `clear(X)` are introduced. The predicate `on` is used to express that block `X` is on `Y`, where `Y` may be either another block or the table. For example, `on(a,b)` is used to represent the fact that block `a` is on block `b` and `on(b,table)` is used to represent that block `b` is on the table. The predicate `clear` is used to represent that nothing is on top of a block and to express that the table is clear, i.e. there is always an empty spot on the table where a block can be moved to. It is possible to derive that a block is clear from the facts expressed in terms of the `on` predicate and we will introduce a logical rule to do so below. It is not possible to similarly derive that the table is always clear (because it is a basic assumption we have made) and we need to represent this fact explicitly by means of the expression `clear(table)`. Finally, to be able to distinguish blocks from the table, the expression `block(X)` is introduced to express that `X` is a block.
Using the `on` predicate makes it possible to define the states a Blocks World can be in. A *state* is defined as a set of facts of the form `on(X,Y)` that is consistent with the basic "laws" of the Blocks World introduced above. Assuming that the set of blocks is given, a state that contains a fact `on(X,Y)` for each block `X` in that set is called *complete*, otherwise it is called a *partial* state. In the remainder, we only consider complete states. It is now also possible to formally define a Blocks World problem. A Blocks World problem is a pair $\langle B_{initial}, G\rangle$ where $B_{initial}$ denotes

[5] It is not always possible to make a constructive move, which explains why it is sometimes hard to solve a Blocks World problem optimally. In that case the state of the Blocks World is said to be in a *deadlock*, see [40] for a detailed explanation.

the initial state and G denotes the goal state. The labels $B_{initial}$ and G have been intentionally used here to indicate that the set of facts that represent the initial state correspond with the initial beliefs and the set of facts that represent the goal state correspond with the goal of an agent that has as its main aim to solve a Blocks World problem.

```
main BlocksWorldAgent
{ This agent solves the Blocks World problem of Figure 1.
   knowledge{
     block(a), block(b), block(c), block(d), block(e), block(f), block(g).
     clear(table).
     clear(X) :- block(X), not(on(Y,X)).
     tower([X]) :- on(X,table).
     tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
   }
   beliefs{
     on(a,b), on(b,c), on(c,table), on(d,e), on(e,table), on(f,g), on(g,table).
   }
   goals{
     on(a,e), on(b,table), on(c,table), on(d,c), on(e,b), on(f,d), on(g,table).
   }
   program{
     if a-goal(tower([X,Y|T])), bel(tower([Y|T])) then move(X,Y).
     if a-goal(tower([X|T])) then move(X,table).
   }
   action-spec{
     move(X,Y) {
        pre{ clear(X), clear(Y), on(X,Z) }
        post{ not(on(X,Z)), on(X,Y) }
     }
   }
}
```

Table 4.2 GOAL Agent Program for solving the Blocks World Problem of Figure 4.1

In the agent program listed in Table 4.2 the **beliefs** section consists of the facts that represent the initial state of the Blocks World problem of Figure 4.1. These facts are represented in the program as a single conjunction (where the comma-symbol denotes conjunction in Prolog). It would not have made a difference if each of these facts would have been represented as individual clauses separated here by the period-symbol. Similarly, the goal state corresponding with Figure 4.1 is represented as a single conjunction in the **goals** section in the program. In the **goals** section, however, it is important to represent the goal to be achieved as a single conjunction. The reason is that each of the facts present in the **goals** section need to be achieved *simultaneously*. If these facts would have been included as clauses separated by the period-symbol this would have indicated that the agent has *multiple, independent goals*. Observe that it is not the same to have two separate goals `on(a,b)` and `on(b,c)` instead of a single goal `on(a,b), on(b,c)` as in the first case we may put `a` on top of `b`, remove `a` again from `b`, and put `b` on top of `c` which would not achieve a state where `a` is on top of `b` which is

on top of `c` simultaneously.[6] It thus is important to keep in mind that, from a logical point of view, the period-symbol separator in the **beliefs** (and **knowledge** section) means the same as the conjunction operator represented by the comma-symbol, but that the meaning of these separators is different in the **goals** section. In the **goals** section the conjunction operator is used to indicate that facts are part of a *single* goal whereas the period-symbol separator is used to represent that an agent has *several different* goals that need not be achieved simultaneously. As separate goals may be achieved at different times it is also allowed that single goals when they are taken together are *inconsistent*, where this is not allowed in the **beliefs** section of an agent. For example, an agent might have the two goals `on(a,b)` and `on(b,a)`. Obviously these cannot be achieved simultaneously, but they can be achieved one after the other.
Facts that may change at runtime should be put in the **beliefs** section. They are used to initialise the belief base of a GOAL agent that may change when a GOAL agent performs actions. Facts that do not change may be put in the **knowledge** section of a GOAL agent. These are used to initialise the knowledge base of the agent which is never modified at runtime. For this reason, the facts of the form `block(X)` representing the blocks present in the Blocks World are put in the **knowledge** section in Table 4.2. All blocks present in Figure 4.1 are enumerated in this section. The fact that the table is clear is also put in the **knowledge** section. In addition, domain knowledge related to the Blocks World is represented here. For example, the rule `clear(X) :- block(X), not(on(Y,X))` can be read as defining when a block `X` is clear, which is the case whenever there is no other block on top of `X`. Observe that this rule is only correct if a state represented by the agent's beliefs is complete, as the negation of Prolog succeeds whenever no proof can be constructed for `on(Y,X)` (*negation as failure*). That is, Prolog supports the *closed world assumption* which is the presumption that what is not currently known to be true is false.
A GOAL agent derives conclusions by combining its knowledge and beliefs. This allows an agent to draw conclusions about the current state it believes it is in using the rules present in the **knowledge** section. For example, the agent in Table 4.2 may derive that `clear(a)`, which expresses that block `a` is clear, by means of the rule `clear(X) :- block(X), not(on(Y,X))`. This follows since we have `block(a)` according to the knowledge base of the agent and the belief base does not contain a fact `on(X,a)` for any `X`.
Although a programmer may also include rules in the **beliefs** section it is a better practice to include these in the **knowledge** section. One reason is that GOAL does not allow to modify such rules at runtime. Another reason is that rules present in the **knowledge** section may also be used when reasoning with goals. The definition of the predicate `tower` in the **knowledge** section in Table 4.2 provides

[6] Incidentally, note that these observations are related to the famous *Sussman anomaly*. Early planners were not able to solve simple Blocks World problems because they constructed plans for subgoals (parts of the larger goal) that could not be combined into a plan to achieve the main goal. The Sussman anomaly provides an example of a Blocks World problem that such planners could not solve, see e.g. [17].

an example where this is useful. The rules that define this predicate define when a list of blocks `[X|T]` is a tower. The first rule `tower([X]) :- on(X,table)` requires that the basis of a tower is grounded on the table. The second rule recursively defines that whenever `[Y|T]` is a tower, extending this tower with a block `X` on top of `Y` also yields a tower, that is, `[X,Y|T]` is a tower. Observe that it is not required that block `X` is clear and a stack of blocks that is part of a larger tower also is considered to be a tower. For example, it is possible to derive `tower([b,c])` using the facts representing the initial state depicted in Figure 4.1.
It turns out that the concept of a tower is particularly useful for defining when a block is in position or misplaced. In order to provide such a definition, however, we need to be able to derive that an agent has the goal of realizing a particular tower. This cannot be derived from the information present in the goal base of the example agent but requires additional conceptual knowledge which defines the notion of a tower. In combination with the conceptual knowledge present in the knowledge base it is possible, however, to derive such a goal, which illustrates that it is useful to derive conclusions from a *single* goal in combination with the knowledge base. By doing so, for example, it is possible for the example agent of Table 4.2 to derive that `tower([e,b])` is a (sub)goal. It can do so by means of the rules that define the predicate `tower` in the knowledge base of the agent and the (sub)goals `on(b,table)` and `on(e,b)` in the goal base.

Mental State Conditions

Agents that derive their choice of action from their beliefs and goals need the ability to inspect their mental state. In GOAL, *mental state conditions* provide the means to do so. These conditions are used in action rules to determine which actions the agent may consider to perform. A mental state condition consists of mental atoms which are conditions on the belief base of the form `bel(`φ`)` and conditions on the goal base of the form `goal(`φ`)` where φ is a conjunction of literals.
Informally, `bel(`φ`)` can be read as "the agent believes that φ". `bel(`φ`)` holds whenever φ can be derived from the belief base *in combination with the knowledge base*. Using the same example as above, it follows in the initial state that `bel(clear(a))`, which expresses that the agent believes that block `a` is clear. Similarly, `goal(`φ`)` can be read as "the agent has a goal that φ". `goal(`φ`)` holds whenever φ can be derived from *a single goal* in the goal base *in combination with the knowledge base*.[7] Again using the same example as above, it follows given the goal base of Table 4.2 and the definition of the `tower` predicate in the **knowledge** section that `goal(tower([e,b]))` since `on(b,table)` and `on(e,b)` are present in the goal base.

[7] This reading differs from that provided in [3] where the goal operator is used to denote *achievement goals*, which additionally require that the agent *does not believe that* φ. The goal operator `goal` introduced here is more basic and, in combination with the belief operator `bel`, allows to define achievement goals.

A mental state condition is a conjunction of mental atoms of the form `bel(`φ`)` and `goal(`φ`)`, or a negation of a mental state condition ψ of the form `not(`ψ`)`. For example, the mental state condition
`goal(on(b,table)), not(bel(on(b,table))`
expresses that the agent has a goal that block `b` is on the table but does not believe that this is the case (yet). Such goals that have still to be achieved are also called *achievement goals*. As achievement goals are important reasons for choosing actions and are frequently used in GOAL programs to this end, a new operator `a-goal(`φ`)` is introduced as an abbreviation for mental state conditions of the form `goal(`φ`), not(bel(`φ`))`.[8]

$$\texttt{a-goal(}\varphi\texttt{)} \overset{df}{=} \texttt{goal(}\varphi\texttt{), not(bel(}\varphi\texttt{))}$$

Interestingly, this operator provides what is needed to express that a block is misplaced as a block is misplaced whenever the agent believes that the block's current position is different from the position the agent wants it to be in.[9] As the position of the tower which a block is part of is irrelevant, the fact that a block `X` is not in position can be represented by `a-goal(tower([X|T]))` where `T` is a tower. `a-goal(tower([X|T]))` expresses that in the goal state block `X` is on top of the tower `T` but the agent does not believe that this is already so in the current state. The concept of a misplaced block is important for defining a strategy to resolve a Blocks World problem, since only misplaced blocks have to be moved, and can be expressed easily and elegantly in GOAL using mental state conditions. Another useful mental state condition is `goal(`φ`), bel(`φ`)` which expresses that a (sub)goal has been achieved. Instantiating the template φ with `tower([X|T])`, this condition expresses that the current position of a block `X` corresponds with the position it has in the goal state.[10] In this case φ is a (sub)goal that is achieved and we call such a (sub)goal a *goal achieved*. The operator `goal-a(`φ`)` is introduced as an abbreviation to denote such goals.

$$\texttt{goal-a(}\varphi\texttt{)} \overset{df}{=} \texttt{goal(}\varphi\texttt{), bel(}\varphi\texttt{)}$$

The condition `a-goal(tower([X,Y|T])), bel(tower([Y|T])` provides another useful example of a mental state condition. It expresses that the achievement goal to construct a tower `tower([X,Y|T]))` has been realized except for the fact that block `X` is not yet on top of tower `[Y|T]`. It is clear that whenever it is possible to move block `X` on top of block `Y` the agent would get closer to achiev-

[8] See [20] for a discussion of this definition.

[9] Actually, here the difference between *knowledge* and *belief* is important as we normally would say something is misplaced only if we *know* that the block is in a different position. That is, an agent's beliefs about the block's position must also correspond with the actual position of the block. If, in fact, the block is in the desired position, in ordinary language, we would say that the block is *believed to be misplaced* but that in fact it is not.

[10] Note that it would not be possible to express this using an achievement goal operator. In [21] the `goal-a` operator is used to define the concept of a deadlock [40].

ing (one of) its goals. Such a move, moreover, would be a *constructive move* which means that the block would never have to be moved again. As the possibility to make a move may be verified by checking whether the precondition of the move action holds (see below), in combination with the mental state condition, we are able to verify whether a constructive move can be made. This condition therefore is very useful to define a strategy for solving Blocks World problems, and is used in the first action rule in the **program** section listed in Table 4.2.

Actions

In order to achieve its goals an agent needs to select and perform actions. Unlike other programming languages, but similar to planners, actions that may be performed by a GOAL agent need to be specified by the programmer of that agent. GOAL does provide some special built-in actions but typically most actions that an agent may perform are derived from the environment that the agent acts in. Actions are specified in the **action-spec** section of a GOAL agent. These actions are called *user-defined actions*. Actions are specified by specifying the conditions when an action can be performed and the effects of performing the action. The former are also called *preconditions* and the latter are also called *postconditions*. The **action-spec** section consists of a set of STRIPS-style specifications [27] of the form (cf. Table 4.1):

```
action{
   pre{precondition}
   post{postcondition}
}
```

The *action* specifies the *name* of the action and its *arguments or parameters* and is of the form *id[args]*, where *id* denotes the name of the action and the *[args]* part denotes an optional list of parameters of the form $(p_1, \ldots, p_n)$, where the p_i are Prolog terms. If an agent is connected to an environment, the user-defined actions will be sent to the environment for execution. (In that case it is important that the name of an action corresponds with the name the environment expects to receive when it is requested to perform the action.) The parameters of an action in a GOAL agent may contain free variables which are instantiated at runtime. An action can only be performed if *all free variables in parameters of an action as well as in the postcondition of the action have been completely instantiated*. This is not only true for user-defined actions but also for built-in actions.
The *precondition* in an action specification is a conjunction of literals. Preconditions are used to verify whether it is possible to perform an action. A precondition φ is evaluated by verifying whether (an instantiation of) φ can be derived from the belief base (as always, in combination with knowledge in the knowledge base). Any free variables in a precondition may be instantiated during this process just like executing a Prolog program returns instantiations of variables. An action is said to be *enabled* whenever its precondition is believed to be the case by the agent.

A *postcondition* specifies the effect of an action. A postcondition is a conjunction of literals. In GOAL effects of an action are changes to the mental state of an agent. The effect φ of an action is used to update the beliefs of the agent to ensure the agent believes φ after performing the action. In line with STRIPS terminology, a *postcondition* φ is also called an *add/delete list* (see also [17, 27]). Positive literals φ in a postcondition are said to be part of the add list whereas negative literals `not(`φ`)` are said to be part of the delete list. The effect of performing an action is that it updates the belief base by first removing all facts φ present in the delete list and thereafter adding all facts present in the add list. Finally, as an action can only be performed when all free variables in the postcondition have been instantiated, each variable present in a postcondition must also be present in the action parameters or precondition of the action.

In addition, performing an action may affect the goal base of an agent. As a rational agent should not invest resources such as energy or time into achieving a goal that has been realized, such goals are removed from the goal base. That is, goals in the goal base that have been achieved as a result of performing an action are removed. Goals are removed from the goal base, however, only if they have been *completely* achieved. The idea here is that a goal φ in the goal base is achieved only when all of its subgoals are achieved. An agent should not drop any of these subgoals before achieving the overall goal as this would make it impossible for the agent to ensure the overall goal is achieved at a single moment in time (see also the reference to the Sussman anamoly above). The fact that a goal is only removed when it has been achieved implements a so-called *blind commitment strategy* [34]. Agents should be committed to achieving their goals and should not drop goals without reason. The default strategy for dropping a goal in GOAL is rather strict: only do this when the goal has been completely achieved. This default strategy can be adapted by the programmer for particular goals by using the built-in `drop` action.

In the GOAL agent of Table 4.2 only one action `move(X,Y)` has been specified. The precondition specifies that in order to be able to perform action `move(X,Y)` of moving `X` on top of `Y` both `X` and `Y` have to be clear. In addition, the literal `on(X,Z)` in the precondition retrieves in variable `Z` on which particular thing, i.e. block or table, `X` is currently on, in order to be able to remove this fact after performing the action. The precondition of `move(X,Y)` in Table 4.2 could have been strengthened by including a condition `not(on(X,Y))` to prevent moves which move a block `X` on top of block `Y` in case block `X` already is on top of `Y`. Clearly, such actions are redundant for solving a Blocks World problem. However, as we will see below, such move options are never generated by the action selection mechanism of GOAL given the action rules in the **program** section. It would be useful to include `not(X=Y)`, however, to prevent moving a misplaced block on the table to another place on the table. The postcondition `not(on(X,Z)), on(X,Y)` of the action `move(X,Y)` has the effect of (first) removing the current position `on(X,Z)` of block `X` from the belief base and (thereafter) adding the new position `on(X,Y)` to it. Even though the precondition does not preclude moving a block on top of another block it is already on, observe that in the case that `Z=Y` the belief base would not change as a result of performing the action.

In addition to the possibility of specifying user-defined actions, GOAL provides several built-in actions for changing the beliefs and goals of an agent, and for communicating with other agents. Here we only briefly discuss the two built-in actions `adopt(`φ`)` and `drop(`φ`)` which allow for modifying the goal base of an agent. The action `adopt(`φ`)` is an action to adopt a new goal φ. The precondition of this action is that the agent does not believe that φ is the case, i.e. in order to execute `adopt(`φ`)` we must have `not(bel(`φ`))`. The idea is that it would not be rational to adopt a goal that has already been achieved. The effect of the action is the addition of φ as a single, new goal to the goal base. The action `drop(`φ`)` is an action to drop goals from the goal base of the agent. The precondition of this action is always true and the action can always be performed. The effect of the action is that any goal in the goal base from which φ can be derived is removed from the goal base. For example, the action `drop(on(a,table))` would remove all goals in the goal base that entail `on(a,table)`; in the example agent of Table 4.2 the only goal present in the goal base would be removed by this action.

Action Rules

The **program** section specifies the *strategy* used by the agent to select an action to perform by means of *action rules*. Action rules provide a GOAL agent with the know-how that informs it when it is opportune to perform an action. In line with the fact that GOAL agents derive their choice of action from their beliefs and goals, action rules consist of a mental state condition *msc* and an action *action* and are of the form **if** *msc* **then** *action*. The mental state condition in an action rule determines whether the corresponding action may be considered for execution or not. If (an instantiation of) a mental state condition is true, the corresponding action is said to be *applicable*. Of course, the action may only be executed if it is also *enabled*. If an action is both applicable and enabled we say that it is an *option*. We also say that action rules *generate options*.

The **program** section of Table 4.2 consists of two action rules. These rules specify a simple strategy for solving a Blocks World problem. The rule **if** `a-goal(tower([X,Y|T])), bel(tower([Y|T]))` **then** `move(X,Y)` specifies that `move(X,Y)` may be considered for execution whenever `move(X,Y)` is a constructive move (cf. the discussion about the mental state condition of this rule above). The rule **if** `a-goal(tower([X|T]))` **then** `move(X,table)` specifies that the action `move(X,table)` of moving block `X` to the table may be considered for execution if the block is misplaced. As these are all the rules, the agent will only generate options that are constructive moves or move misplaced blocks to the table, and the reader is invited to verify that the agent will never consider moving a block that is in position or making a redundant move that puts a block on top of a block that it already is on. Furthermore, observe that the mental state condition of the second rule is weaker than the first. In common expert systems terminology, the first rule *subsumes* the second as it is more specific.[11] This implies that whenever a constructive move `move(X,Y)` is an option the action `move(X,table)` is also an option. The set of options generated by the action rules thus may consist of more than one action. In that case, GOAL *arbitrarily* selects one action out of the set of all options. As a result, a GOAL agent is nondeterministic and may execute differently each time it is run. A set of action rules may be viewed as specifying a *policy*. There are two differences with standard definitions of a policy in the planning literature, however [17]. First, action rules do not need to generate options for each possible state. Second, action rules may generate *multiple* options in a particular state and do not necessarily define a function from the (mental) state of an agent to an action. A policy for a GOAL agent thus does not need to be *universal*[12] and may *underspecify* the choice of action of an agent.

Execution Traces of The Blocks World Agent

We will trace one particular execution of the Blocks World agent of Table 4.2 in more detail here. As a GOAL agent selects an arbitrary action when there are more options available, there are multiple traces that may be generated by the agent, three of which are listed below.

In the initial state, depicted also in Figure 4.1, the agent can move each of the clear blocks `a`, `d`, and `f` to the table. It is easy to **verify the precondition** of the `move` action in each of these cases by instantiating the action specification of the `move` action and inspecting the knowledge and belief bases. For example, instantiating `move(X,Y)` with block `a` for variable `X` and `table` for variable `Y` gives the corresponding precondition `clear(a), clear(table), on(a,Z)`. By inspection of the knowledge and belief bases, it immediately follows that `clear(table)`, and we find that by instantiating variable `Z` with `b` it follows that `on(a,Z)`. Using the rule for `clear` it also follows that `clear(a)` and we conclude that action `move(a,table)` is enabled. Similar reasoning shows that the actions `move(d,table)`, `move(f,table)`, `move(a,d)`, `move(a,f)`, `move(d,a)`, `move(d,f)`, `move(f,d)`, `move(f,a)` are enabled as well. The reader is invited to check that no other actions are enabled.

(continued overleaf)

[11] Thanks to Jörg Müller for pointing this out.

[12] In the sense of [37], where a "universal plan" or policy specifies the appropriate action for *every* possible situation.

A GOAL agent selects an action using its action rules. In order to verify whether moving the blocks `a`, `d`, and `f` to the table are options we need to **verify applicability** of actions by checking the mental state conditions of action rules that may generate these actions. We will do so for block `a` here but the other cases are similar. Both rules in the program section of Table 4.2 can be instantiated such that the action of the rule matches with `move(a,table)`. As we know that block `a` cannot be moved constructively, however, and the mental state condition of the first rule only allows the selection of such constructive moves, this rule is not applicable. The mental state condition of the second rule expresses that a block `X` is misplaced. As block `a` clearly is misplaced, this rule is applicable. The reader is invited to verify this by checking that `a-goal([a,e,b])` holds in the initial state of the agent.

Assuming that `move(a,table)` is selected from the set of options, the action is executed by **updating the belief base** with the instantiated postcondition `not(on(a,b)), on(a,table)`. This means that the fact `on(a,b)` is removed from the belief base and `on(a,table)` is added. The goal base may need to be updated also when one of the goals has been completely achieved, which is not the case here. As in our example, we have abstracted from perceptions, there is no need to process any and we can repeat the action selection process again to select the next action.

As all blocks except for blocks `c` and `g` are misplaced, similar reasoning would result in a possible trace where consecutively `move(b,table)`, `move(d,table)`, `move(f,table)` are executed. At that point in time, all blocks are on the table, and the first rule of the program can be applied to build the goal configuration, e.g. by executing `move(e,b)`, `move(a,e)`, `move(d,c)`, `move(f,d)`. In this particular trace the goal state would be reached after performing 8 actions.

Additionally, we list the 3 shortest traces - each including 6 actions - that can be generated by the Blocks World agent to reach the goal state:

$$\begin{array}{l}\text{Trace1}: \texttt{move}(\texttt{a},\texttt{table}),\texttt{move}(\texttt{b},\texttt{table}),\texttt{move}(\texttt{d},\texttt{c}),\texttt{move}(\texttt{f},\texttt{d}),\texttt{move}(\texttt{e},\texttt{b}),\texttt{move}(\texttt{a},\texttt{e}).\\ \text{Trace2}: \texttt{move}(\texttt{a},\texttt{table}),\texttt{move}(\texttt{b},\texttt{table}),\texttt{move}(\texttt{d},\texttt{c}),\texttt{move}(\texttt{e},\texttt{b}),\texttt{move}(\texttt{f},\texttt{d}),\texttt{move}(\texttt{a},\texttt{e}).\\ \text{Trace3}: \texttt{move}(\texttt{a},\texttt{table}),\texttt{move}(\texttt{b},\texttt{table}),\texttt{move}(\texttt{d},\texttt{c}),\texttt{move}(\texttt{e},\texttt{b}),\texttt{move}(\texttt{a},\texttt{e}),\texttt{move}(\texttt{f},\texttt{d}).\end{array}$$

There are many more possible traces, e.g. by starting with moving block `f` to the table, all of which consist of more than 6 actions.

To conclude the discussion of the example Blocks World agent, in Figure 4.2 the RSG line shows the average performance of the GOAL agent in number of moves relative to the number of blocks present in a Blocks World problem. This performance is somewhat better than the performance of the simple strategy of first moving all blocks to the table and then restacking the blocks to realize the goal state indicated by the US line[13] as the GOAL agent may perform constructive

[13] Observe that this simple strategy never requires more than $2N$ moves if N is the number of blocks. The label "US" stands for "Unstack Strategy" and the label "RSG" stands for "Random Select GOAL", which refers to the default action selection mechanism used by GOAL.

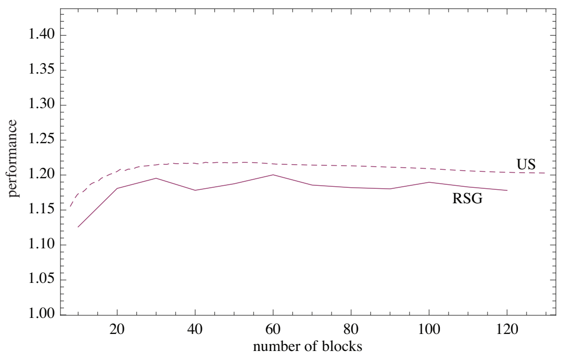


Fig. 4.2 Average Performance of a Blocks World GOAL Agent

moves whenever this is possible and not only after moving all blocks to the table first.

4.2.1.2 Modules and Focus of Attention

Rational agents are assumed to create partial plans for execution and to not over-commit to a particular way of achieving a goal. One important reason for not computing a complete plan is that in a dynamic, uncertain environment an agent typically does not have sufficient knowledge to fill in the details of a plan that is guaranteed to succeed. It therefore is better practice to decide on the action to perform next when the required information is available. As the action selection mechanism in GOAL ensures that agents select their actions by inspection of their current mental state overcommitment is avoided. As a result, the Blocks World agent, for example, provides a robust solution for solving Blocks World problems because it is flexible in its choice of action. It would still perform well even if other agents would interfere, assuming that it is able to perceive what happens in the Blocks World.

Even though action rules provide for a flexible choice of action it is useful to add additional structure to a GOAL agent. As is the case in almost any programming language, it is useful to be able to structure parts of a program in a single unit. In GOAL it is useful to combine related conceptual and domain knowledge, goals, actions and action rules that are relevant for handling particular situations in a single structure. *Modules* provide such a structure in GOAL. Modules provide for reusability and the encapsulation of related knowledge, goals, actions and action rules. They also provide a programmer with a tool to focus on the particular knowledge and skills that an agent needs to handle a situation.

Modules in GOAL also provide for focus in another sense. In many situations it is natural to *focus attention on achieving particular goals* and disregard other goals for the moment. Such focus allows for a more dedicated use of resources and the need for creating plans for a subset of ones goals only. It also allows for sequencing potentially conflicting goals. As an example, consider a truck delivery domain where a truck is supposed to deliver multiple packages to different locations. Given that the load of packages that the truck may carry is limited, it is useful to focus on the delivery of packages to a particular subset of locations and only load packages that need to be delivered to those locations. Modules provide for a mechanism that enables agents to focus attention in this way. In the remainder we will illustrate the use of modules in the Blocks World domain. This example provides a simple illustration of programming with modules which also illustrates how modules can be used to program a different strategy for solving a Blocks World problem.

Syntactically, a module is very similar to a GOAL agent. The main difference with a GOAL agent such as the Blocks World agent discussed in the previous section is that a module has an additional `Context` section, which specifies an *activation condition*. A distinguishing feature of modules in GOAL is that the context of a module is specified *declaratively*. A module's context specifies not only when to activate the module but also for what purpose a module is activated. It thus provides a declarative specification of the intended use of a module. Such specifications are useful for a programmer as a programmer does not have to inspect the implementation details inside a module but can read off the intended use from the context.

Another difference with a GOAL agent is that a module does not have a **beliefs** section and that all sections other than the **program** section are optional. The reason that a module does not have a **beliefs** section is that a module specifies knowledge and skills that are independent of the current state. A module specifies the generic knowledge and know-how to deal with a particular situation but not the specifics of a particular state. The belief base of an agent is used to keep track of the state of the environment and is a "global" component of the agent. This means that the beliefs of an agent are accessible by and may be modified by any module. The **knowledge** and **action-spec** section are optional because the knowledge in the **knowledge** section and all actions specified in the **action-spec** section of the GOAL agent that contains the module are "inherited" by the module and are "globally" accessible as beliefs are. The same does not hold for the goals of an agent, however. The context of a module provides a filter on the set of goals that the agent currently pursues which allows an agent to focus its attention on a subset of these goals.

Table 4.3 presents an example module, which can be used by the Blocks World agent introduced above. In the remainder we assume that the action rules used by the original agent of Table 4.2 are replaced by the module of Table 4.3 and we explain how this change modifies the behaviour of that agent.

The *context* of a module is a mental state condition that serves two functions. The first function is that a context specifies when a module may be activated. For example, the context of the module in Table 4.3 specifies that the module may be activated whenever the agent has an achievement goal to build a tower with block `X` as the top of that tower. That is, block `X` should be clear. The context can also be viewed as a (pre)condition for activating a composed activity, or a policy, as a

```
module BuildTower
{ % This module achieves the goal of building a particular tower of blocks.
   Context{
     a-goal(clear(X), tower([X|T]))
   }
   program{
     if a-goal(tower([X,Y|T])), bel(tower([Y|T])) then move(X,Y).
     if bel(tower([X|T]), not(goal(tower([X|T])) then move(X,table).
   }
   action-spec{
     move(X,Y) {
        pre{ clear(X), clear(Y), on(X,Z) }
        post{ not(on(X,Z)), on(X,Y) }
   }
}
```

Table 4.3 Module Replacing the **program** Section of the Blocks World Agent

set of action rules in a module specifies such a policy. The second function of a context is that it is used as a filter on the goals that the agent pursues which selects a subset of these goals. The goals currently pursued by an agent are said to be in the agent's *attention set*. After activating a module the attention set of an agent is restricted to those goals in that set that are obtained from a particular instantiation of the context of a module. The goals that are put in the updated attention set are all goals φ that are in the current attention set and correspond with a positive occurence of a mental atom `goal(`φ`)` in the instantiated context.[14] This means all other goals in the current attention set of the agent are removed and, that, if a context does not have positive occurrences of such mental atoms all goals in this set are removed.[15] Any goals introduced by the module's **goals** section are added to this updated attention set. For example, upon activation of the module listed in Table 4.3, the context of the module is instantiated such that it becomes true. Assume that the instantiated context is `a-goal(clear(g), tower([g]))`, which is an achievement goal in the initial state of Figure 4.1. As this context is an abbreviation for a mental state condition with a positive occurrence of a mental atom of the form `goal(clear(g), tower([g]))` the goal `clear(g), tower([g])` is included in the attention set of the agent and all other goals are removed from the attention set. As the **goals** section in the module is absent, the resulting attention set would consist of the single goal `clear(g), tower([g])`.

[14] A mental atom `goal(`φ`)` occurs positively in a context if it occurs within the scope of an even number of negations `not`.

[15] Formally, a filter function $filter(c, m)$ with c a context and m a mental state (with a goal base that provides the current attention set) can be defined as follows: $filter(c, m) = \{\varphi \mid m \models_c c\theta \;\&\; \mathtt{goal}(\varphi) \in pos(c\theta)\}$ where $pos(c)$ denotes the set of all positive occurrences of mental atoms in c and θ is a substitution for variables that occur in c. For a definition of the entailment relation $\models_c$ see Section 4.2.2.1. The filter function $filter(c, m)$ provides the new attention set after activating a module with context c.

A module provides not only a means to focus on particular goals but also provides a context which restricts the choice of action. When a module is activated the action rules present in the module are the only rules available to generate action options. A module may also introduce action specifications that are only available while the module is executed and specific for handling situations the module has been designed for. Actions specified in the main GOAL agent, but not those specified in other modules, are also accessible from within a module. In the example in Table 4.3 the `move` action has been moved from the main GOAL agent to the module. As a result, it is only possible to move blocks when the module is activated.

The example module replaces the action rules in the **program** section of the Blocks World agent of Table 4.2. The first action rule of that agent which generates constructive moves is included in the **program** section of the module. The second action rule of this agent which generates moves of misplaced blocks to the table, however, has been replaced by another rule. The reason is that the original rule assumed that each block is part of the goal configuration and, as a consequence, any block is either in position or misplaced. As the attention set of an agent upon activation of a module is restricted we can no longer make this assumption. Instead of being part of a goal condition a block may now be in the way of achieving a goal of the agent, i.e. it may obstruct making moves with a block that is part of such a goal because it is above such a block. Therefore, the second action rule **if** `bel(tower([X|T]), not(goal(tower([X|T]))` **then** `move(X,table)` in Table 4.3 still moves blocks to the table but under a different condition. The mental state condition of this rule expresses that block `X` is possibly in the way to get to a block needed to achieve a goal of the agent. Here, *possibly in the way* means that the agent does not intend the block to be in the position it believes it to be in.[16] Observe that blocks that are misplaced also satisfy this mental state condition but that blocks that are possibly in the way do not always satisfy the mental statement condition
`goal(tower([X|T])), not(bel(tower([X|T])))`. The latter condition expresses that block `X` is misplaced and therefore must be part of the agent's goals whereas a block that is possibly in the way does not need to be part of one of the goals of the agent.[17]

[16] We use "does not intend" here instead of the seemingly more natural "does not want" as the agent is supposed to *not* have a goal here. The natural language expression "does not want φ" is more commonly used to express that one wants to be in a state where φ is *not* the case (the effect of which can be strengthed by putting more stress on "not" in the phrase). In other words, this expression is commonly used to express that one has a goal to achieve that φ is not the case. In contrast, the expression "does not intend" is more commonly used to express the lack of an intention or goal. From a more technical point of view, as the knowledge representation used is Prolog, there is no difference between writing `not(goal(`φ`))` or `goal(not(`φ`))` since in Prolog the Closed World Assumption is supported (a similar point can be made for the `bel` operator). The negation in Prolog is negation as failure and cannot be used to express "explicit" negation which would be needed to make the distinction.

[17] Suppose that block `X` is misplaced and the agent believes that `X` is part of a tower `[X|T]`. In that case, the agent has a goal that the block is part of another tower. That is, we have

The second action rule may generate options that are not needed to realize the achievement goal of the agent as there may be stacks of blocks which do not contain a block needed to build the desired tower and these blocks therefore are not in the way to achieve this goal. The reader is invited to provide a mental state condition that more accurately captures the notion of *a block being in the way*. (Hint: it is useful to introduce a Prolog definition of the concept *above*.) The strategy of building towers in the goal state one by one implemented using the module construct, however, never requires more than $2N$ steps where N is the number of blocks.

Activating a module is making a commitment to achieve the goals in the attention set that is initialised upon activation. A module is terminated only when the attention set, i.e. the set of goals currently pursued by the agent, is empty. The knowledge and skills incorporated in a module need to be sufficient in order to realize the goals in the agent's attention set. In addition, another module may be activated from a module whenever the context of that module is true. In the example, the agent has a goal to achieve `clear(g), tower([g])` and after moving block `f` to the table this goal has been achieved and is removed from the attention set and, as a result, the module is terminated. Upon termination the agent's previous goals except for those that have been *completely* achieved by the module are put back into the attention set and the agent continues execution.[18]

4.2.2 Semantics and Verification

In this section we introduce the formal semantics of GOAL and discuss the verification framework for the language. The semantics of GOAL consists of several more or less independent parts. The first part defines the semantics of the agent's mental state and the mental state conditions that can be used to inspect such states. The second part defines the semantics of actions and the agent's action rules used for choosing an action to perform. The various parts combined together define the *operational semantics* of GOAL.

4.2.2.1 Semantics of Mental States

GOAL is a general-purpose agent programming language. The basic design of the language assumes that beliefs and goals of an agent are specified in a *declarative* way. Beliefs of a GOAL agent thus do not encode procedural knowledge but represent *what is the case* and goals of a GOAL agent do not specify which actions an agent wants to perform but represent *what state an agent wants to achieve*. The

`not(goal([X|T]))`. Vice versa, it is not possible to derive from the fact that a block is possibly in the way that the block is part of one of the goals of the agent and we cannot conclude the block is misplaced.

[18] For further details on and explanation of modules the reader is referred to [19].

main benefit of using declarative specifications to represent an agent's beliefs and goals is that it allows an agent to *reason* with its beliefs and goals. GOAL thus aims to facilitate the design of agent programs at the *knowledge level* [31].

An agent's mental state consists of its knowledge, its beliefs and its goals as explained in Section 4.2.1.1. In the current implementation of GOAL these are represented in Prolog [41, 42]. The knowledge and beliefs of an agent in this implementation are stored in two different Prolog databases; the storage of goals in this implementation is slightly more complicated because of the difference in semantics of goals and beliefs. The details are not important here, however, since the main point we want to make is that GOAL does not commit to any particular *knowledge representation technology*. Instead of Prolog an agent might use variants of logic programming such as Answer Set Programming (ASP; [1]), a database language such as Datalog [7], the Planning Domain Definition Language (PDDL; [17]), or other, similar such languages, or possibly even Bayesian Networks [32]. The only assumption that we will make throughout is that an agent uses a *single* knowledge representation technology to represent its knowledge, beliefs and goals. For some preliminary work on lifting this assumption, we refer the reader to [13].

In order to abstract from the details of any specific knowledge representation technology in the presentation of the semantics of GOAL, we first define abstractly what we mean by a knowledge representation technology. The basic capabilities that we need such a technology to provide are the capability to *represent* states of affairs (which is fundamental), the capability to *store* these representations in a storage facility, the capability to *reason* with them and the capability to *change* the representations present in a storage. These capabilities are similar to some of the functions associated with a knowledge technology as discussed in [15].

The first capability to represent states of affairs is realized by means of a *language*. The only assumptions we make about this language is that it defines what a *formula* is and that it contains a special formula $\bot$. In other words, we assume that a language defines the grammar or syntax of *well-formed formulae*. We write $\varphi \in \mathcal{L}$ to denote that φ is a formula of language $\mathcal{L}$; in particular, we have $\bot \in \mathcal{L}$. Intuitively, we think of a formula as a *sentence* that *expresses that a state of affairs is the case (or not)* similar to declarative sentences in natural language. Although the meaning of formulae of a language is not formally defined, informally, we think of a formula as having a *truth value* and of a formula being *true* or *false* (but other possible truth values such as *undefined* are also allowed). The special formula $\bot$ is assumed to always have the truth value false and is introduced to be able to define when a set of formulae is inconsistent.

The second capability to store representations is formalised here by means of the notion of a *set*. We thus abstract from most implementation details typically associated with this capability. A knowledge, belief and goal base each are represented in the semantics as a set of formulae, or, equivalently, as a subset of a language $\mathcal{L}$. Below we use $\mathcal{D} \subseteq \mathcal{L}$ to denote a knowledge base, $\Sigma \subseteq \mathcal{L}$ to denote a belief base, and $\Gamma \subseteq \mathcal{L}$ to denote a goal base.

The third capability is realized by means of a *consequence relation* (also called *entailment*). A consequence relation defines when a formula follows from ("is a consequence of") a set of formulae. We use $\models$ to denote consequence relations and write $T \models \varphi$ for φ follows from a set of formulae T. For example, a formula φ follows from an agent's belief base Σ whenever we have $\Sigma \models \varphi$. When the special formula $\bot$ follows from a set T we say that T is *inconsistent*; the intuition here is that in that case T is contradictory, something we typically want to avoid. For example, we would like an agent's knowledge and belief base to be consistent. A consequence relation is the formal counterpart of the reasoning capability of an agent in the semantics since it allows an agent to derive and reason with its knowledge, beliefs, and goals.

The fourth and final capability we need is the capability to *update* an agent's beliefs.[19] Recall that an agent's knowledge base is assumed to be static and does not change since it is assumed to represent conceptual and domain knowledge that does not change (see also section 4.2.1). In particular we will need to be able to define how an agent's beliefs change when it performs an action. In order to do so an update operator denoted by $\oplus$ is assumed that updates a set of formulae T with a formula φ. That is, $T \oplus \varphi$ denotes the new set of formulae obtained after updating T with φ. This will enable us in the next section to say that the resulting belief base of updating a belief base Σ with the effect φ of an action is $\Sigma \oplus \varphi$. See section 4.2.1.1 for a concrete, informally defined STRIPS-style operator.

Summarizing, a knowledge representation technology is defined here as a triple $\langle \mathcal{L}, \models, \oplus \rangle$ with $\mathcal{L}$ a language to represent states of affairs, $\models$ a consequence relation that defines when a formula follows from a set of formulae, and $\oplus$ defines how a set of formulae is updated with a given formula.[20] Using our definition of a knowledge representation technology, it is now easy to formally define what a *mental state* of an agent is and to formally define the semantics of *mental state conditions*. We first define a mental state, since it is needed to define the semantics of mental state conditions as well, and then proceeed to discuss mental state conditions.

A mental state consists of an agent's knowledge, its beliefs, and its goals. Each of these are represented using a particular knowledge representation language $\mathcal{L}$. The knowledge, beliefs and goals of a *rational* agent should satisfy some additional constraints that we will call *rationality constraints*. First, we assume that an agent's knowledge as well as its beliefs are consistent. This is a reasonable assumption, which may be debated by philosophers, logicians and psychologists, but makes sense in the context of an agent programming language. We also assume that *individual* goals $\gamma \in \Gamma$ in the goal base of an agent are consistent. It is irrational for an agent to pursue inconsisent goals, which by definition it cannot achieve.

[19] In the setup we use here, we do not need a special capability to update the goal base when an agent comes to believe it has achieved a goal; in that case we simply remove the goal from the goal base, which is a set-theoretic operation; see the next section.

[20] Technically, we would also need to clarify the notion of a *term* which may be used to instantiate a *variable* in order to specify the use of variables in a GOAL agent, but we abstract from such details here.

The reason that we require single goals in a goal base to be consistent but not conjunctions of multiple goals is that we allow an agent to have conflicting goals in its goal base. For example, an agent may want to achieve a state where the light is on but thereafter may want to achieve a state where the light is off again. Here we assume that the language used to express goals is not capable of expressing such *temporal* dimensions of goals and therefore allow an agent to have multiple goals that when viewed as a single goal would be inconsistent. The main reason for allowing contradictory goals thus is not because we believe that the goals of an agent may be inconsistent but because of the (assumed) lack of expressivity of the knowledge representation language used to represent goals here.[21] Finally, an agent is assumed to only have goals which it does not believe to already have been achieved *completely*. Any rational agent should avoid investing resources into achieving something that is already the case. For that reason it should not have any goals that have already been achieved. Note that an agent is allowed but not required to believe that the opposite of what it wants is the case; for example, it may believe the light is on when it wants to have the light off but does not need to believe so to have the goal.

Definition 4.1. *(Mental State)*
A *mental state* is a triple $\langle \mathcal{D}, \Sigma, \Gamma \rangle$ where $\mathcal{D} \subseteq \mathcal{L}$ is called a *knowledge base*, $\Sigma \subseteq \mathcal{L}$ is a *belief base*, and $\Gamma \subseteq \mathcal{L}$ is a *goal base* that satisfy the following *rationality constraints*:

- An agent's knowledge combined with its beliefs is consistent:

$$\mathcal{D} \cup \Sigma \not\models \bot$$

- Individual goals are consistent with an agent's knowledge:

$$\forall \gamma \in \Gamma : \mathcal{D} \cup \{\gamma\} \not\models \bot$$

- An agent does not have goals it believes to be completely achieved: [22]

$$\forall \gamma \in \Gamma : \mathcal{D} \cup \Sigma \not\models \gamma$$

The next step in defining the semantics of GOAL is to define the semantics of *mental state conditions*. An agent needs to be able to inspect its mental state, and

[21] See for work on extending GOAL with temporal logic as a knowledge representation language [20, 23].

[22] The precise formulation of the rationality constraints relating the contents of the goal base to that of the knowledge and/or belief base of an agent may depend on the knowledge representation language. In particular, when the knowledge representation language allows for expressing *temporal conditions*, e.g. allows for expressing that a state of affairs holds at some time in the future, then these constraints and the semantics of the **G** operator below would be in need of reformulation (see [24]). In that case, the third rationality constraint also could be refined and in addition we could require that an agent should not have any goals it believes are impossible to achieve (a condition which can only be properly expressed using temporal operators).

mental state conditions allow an agent to do so. Mental state conditions are conditions on the mental state of an agent, expressing that an agent believes something is the case, has a particular goal, or a combination of the two (see also section 4.2.1). Special operators to inspect the belief base of an agent, we use $\mathbf{B}(\varphi)$ here, and to inspect the goal base of an agent, we use $\mathbf{G}(\varphi)$ here, are introduced to do so. We allow boolean combinations of these basic conditions but do not allow the nesting of operators. Basic conditions may be combined into a conjunction by means of $\wedge$ and negated by means of $\neg$. For example, $\mathbf{G}(\varphi) \wedge \neg\mathbf{B}(\varphi)$ with $\varphi \in \mathcal{L}$ is a mental state condition, but $\mathbf{B}(\mathbf{G}(\varphi))$ which has nested operators is not. Note that we do not assume the operators $\wedge$ and $\neg$ to be present in the $\mathcal{L}$, and if so, a negation operator might still have a different meaning in $\mathcal{L}$.

Definition 4.2. *(Syntax of Mental State Conditions)*
A mental state condition ψ is defined by the following rules:

$$\begin{array}{rl} \varphi ::= & \text{any element from } \mathcal{L} \\ \psi ::= & \mathbf{B}(\varphi) \mid \mathbf{G}(\varphi) \mid \psi \wedge \psi \mid \neg\psi \end{array}$$

The meaning of a mental state condition is defined by means of the mental state of an agent. A belief condition $\mathbf{B}(\varphi)$ is true whenever φ follows from the belief base combined with the knowledge stored in the agent's knowledge base (in order to define this the consequence relation of the knowledge representation technology is used). The meaning of a goal condition $\mathbf{G}(\varphi)$ is different from that of a belief condition. Instead of simply defining $\mathbf{G}(\varphi)$ to be true whenever φ follows from *all* of the agent's goals (combined with the knowledge in the knowledge base), we will define $\mathbf{G}(\varphi)$ to be true whenever φ follows from *one* of the agent's goals (and the agent's knowledge). This is in line with the remarks above that a goal base may be inconsistent, i.e. may contain multiple goals that taken together are inconsistent. We do not want an agent to conclude it has the absurd goal $\bot$ (i.e. to achieve the impossible). Since individual goals are assumed to be consistent, we can use these individual goals to infer the goals of an agent.

Definition 4.3. *(Semantics of Mental State Conditions)*
Let $m = \langle \mathcal{D}, \Sigma, \Gamma \rangle$ be a mental state. The semantics of mental state conditions ψ is defined by the following semantic clauses:

$$\begin{array}{lll} m \models_c \mathbf{B}(\varphi) & \text{iff} & \mathcal{D} \cup \Sigma \models \varphi, \\ m \models_c \mathbf{G}(\varphi) & \text{iff} & \exists \gamma \in \Gamma : \mathcal{D} \cup \{\gamma\} \models \varphi, \\ m \models_c \psi_1 \wedge \psi_2 & \text{iff} & m \models_c \psi_1 \text{ and } m \models_c \psi_2, \\ m \models_c \neg\psi & \text{iff} & m \not\models_c \psi. \end{array}$$

Note that in the definition of the semantics of mental state conditions we have been careful to distinguish between the consequence relation that is defined, denoted by $\models_c$, and the consequence relation $\models$ assumed to be given by the knowledge representation technology and used to define $\models_c$. The definition thus shows how the meaning of a mental state condition can be derived from the semantics

of more basic notions defined in an arbitrary knowledge representation technology.[23]
In the remainder of this section, it is useful to assume that the knowledge representation language at least provides the propositional operators for conjunction and negation. Here we will simply use the same notation $\wedge$ and $\neg$ also used for mental state conditions to refer to these operators in the knowledge representation language $\mathcal{L}$ as well. Given this assumption, note that because of the fact that a goal base may contain multiple goals that are inconsistent when taken together, it follows that we may have that $\mathbf{G}(\varphi)$ as well as $\mathbf{G}(\neg\varphi)$. It should be clear from our previous discussion however that it does not follow from this that $\mathbf{G}(\varphi \wedge \neg\varphi)$ also holds. To repeat, intuitively, $\mathbf{G}(\varphi)$ should be interpreted as expressing that the agent wants to achieve φ some time in the future. Given this reading of $\mathbf{G}(\varphi)$ it is perfectly consistent for an agent to also have a goal $\neg\varphi$, i.e. $\mathbf{G}(\neg\varphi)$.

P1	if ψ is an instantiation of a classical tautology, then $\models_c \psi$.
P2	if $\models \varphi$, then $\models_c \mathbf{B}\varphi$.
P3	$\models_c \mathbf{B}(\varphi \rightarrow \varphi') \rightarrow (\mathbf{B}\varphi \rightarrow \mathbf{B}\varphi')$.
P4	$\models_c \neg\mathbf{B}\bot$.
P5	$\models_c \neg\mathbf{G}\bot$.
P6	if $\models \varphi \rightarrow \varphi'$, then $\models_c \mathbf{G}\varphi \rightarrow \mathbf{G}\varphi'$.

Table 4.4 Properties of Beliefs and Goals

Some other properties of the belief and goal modalities and the relation between these operators are listed in Table 4.4. Here, we use $\rightarrow$ to denote implication, which can be defined in the usual way by means of the conjunction $\wedge$ and negation $\neg$. The first property (P1) below states that mental state conditions that instantiate classical tautologies, e.g. $\mathbf{B}\varphi \vee \neg\mathbf{B}\varphi$ and $\mathbf{G}\varphi \rightarrow (\mathbf{B}\varphi' \rightarrow \mathbf{G}\varphi)$, are valid with respect to $\models_c$. Property (P2) corresponds with the usual necessitation rule of modal logic and states that an agent believes all validities of the base logic. (P3) expresses that the belief modality distributes over implication. This implies that the beliefs of an agent are closed under logical consequence. Finally, (P4) states that the beliefs of an agent are consistent. In essence, the belief operator thus satisfies the properties of the system KD (see e.g. [30]). Although in its current presentation, it is not allowed to nest belief or goal operators in mental state conditions in GOAL, from [30], section 1.7, we conclude that we may assume as if our agent has positive ($\mathbf{B}\varphi \rightarrow \mathbf{B}\mathbf{B}\varphi$) and negative ($\neg\mathbf{B}\varphi \rightarrow \mathbf{B}\neg\mathbf{B}\varphi$) introspective properties: every formula in the system KD45 (which is KD together with the two mentioned properties) is equivalent to a formula without nestings of operators. Property (P5) states that an agent also does not have inconsistent goals, that is, we have $\models_c \neg\mathbf{G}\bot$. Property (P6) states that the goal operator is closed under implica-

[23] This semantics was first introduced in [22]. For a discussion of alternative semantics for goals, see also [35].

tion in the base language. That is, whenever $\varphi \rightarrow \varphi'$ is valid in the base language then we also have that $\mathbf{G}\varphi$ implies $\mathbf{G}\varphi'$. This is a difference with the presentation in [3] which is due to the more basic goal modality we have introduced here. For the same reason we also have that $\mathbf{B}\varphi \wedge \mathbf{G}\varphi$ is not inconsistent.
We may now put our definitions to work and provide some examples of what we can do. First, as discussed in section 4.2.1, we can introduce some useful abbrevations. In particular, we can define the notions of an *achievement goal* **A-goal**(φ) and the notion of a *goal achieved* **goal-A**(φ) as follows:

$$\begin{aligned} \mathbf{A\text{-}goal}(\varphi) &\overset{df}{=} \mathbf{G}(\varphi) \wedge \neg \mathbf{B}(\varphi), \\ \mathbf{goal\text{-}A}(\varphi) &\overset{df}{=} \mathbf{G}(\varphi) \wedge \mathbf{B}(\varphi). \end{aligned}$$

For some properties of the **A-goal** operator we refer the reader to [3], Lemma 2.4. Both of these defined operators are useful when writing agent programs. The first is useful to derive whether a part of a goal has not yet been (believed to be) achieved whereas the second is useful to derive whether a part of a goal has already been (believed to be) achieved. For some concrete examples, please refer back to section 4.2.1. It should be noted that an agent is allowed to believe part of one of its goals has been achieved but cannot believe that one of its goals has been *completely* achieved as such goals are removed automatically from the goal base. That is, whenever we have $\gamma \in \Gamma$ we must have both **A-goal**(γ) as well as **G**(γ) since it is not allowed by the third rationality constraint in Definition 4.1 that an agent believes γ in that case.
Note that in this section we have only used the first two components of a knowledge representation technology, the language $\mathcal{L}$ and consequence relation $\models$, so far. We will use the third component, the update operator $\oplus$, in the next section to formally define the effects of performing an action.

4.2.2.2 Semantics of Actions and Action Selection

GOAL has a formal, operational semantics defined by means of Plotkin-style transition semantics [33]. The details of the semantics of modules and communication are not discussed here.[24]
In order to define the semantics of actions, we need to model both when an action can be performed as well as what the effects of performing an action are. As actions, except for the built-in actions, are user-defined, we introduce some assumptions about what information is available to define the semantics. First, we assume that it is known which actions the agent can perform, i.e. those actions specified by the programmer in the agent program, and that these actions are given by a set $\mathcal{A}$. Second, we assume that two mappings *pre* and *post*

[24] The reader is referred to [19] for a detailed semantics of modules. Communication in the current implementation of GOAL implements a simple "mailbox semantics" as in 2APL [12]. In GOAL, messages are stored in an agent's mailbox and may be inspected by querying special, reserved predicates `sent` and `received`. See for a discussion also section 4.2.4.

which map actions a from this set of actions $\mathcal{A}$ and mental states m to a formula φ in the knowledge representation language $\mathcal{L}$ are given. The mappings pre and $post$ are assumed to provide the preconditions respectively postconditions associated with an action in a given state. For example, we would have $pre($`move(a,table)`$,m)=$`clear(a), clear(table), on(a,b)` in the initial state mental m of the GOAL agent listed in Table 4.2 and $post($`move(a,b)`$,m)=$`not(on(a,b)),on(a,table)`. Finally, we also assume that the postconditions specified by *post* for each action are consistent with the domain knowledge of the agent. As the domain knowledge of an agent is assumed to be static, it would not be possible to perform an action with a postcondition that conflicts with the agent's domain knowledge without violating the rationality constraints introduced earlier.
The precondition of an action is used to represent when an action can be performed, whereas the postcondition is used to represent the effects of an action. An action may affect both the beliefs and goals of an agent. The postcondition expresses how the beliefs of an agent's mental state should be updated. This is where the update operator $\oplus$ of the knowledge representation technology is useful. The new belief base that results from performing an action $\mathsf{a} \in \mathcal{A}$ can be obtained by applying this operator. In addition, the goals that have been completely achieved need to be removed from the goal base. This transformation of the mental state is formally defined by means of a mental state transformer function $\mathcal{M}$, which also provides the semantics of the built-in actions **adopt** and **drop** below.

Definition 4.4. *(Mental State Transformer $\mathcal{M}$)*
Let pre and $post$ be mappings from $\mathcal{A}$ to $\mathcal{L}$. Then the *mental state transformer function* $\mathcal{M}$ is defined as a mapping from user-defined and built-in actions $\mathcal{A} \cup \{\mathbf{adopt}(\varphi), \mathbf{drop}(\varphi) \mid \varphi \in \mathcal{L}\}$ and mental states $m = \langle \mathcal{D}, \Sigma, \Gamma \rangle$ to mental states as follows:

$$\begin{array}{ll}
\mathcal{M}(\mathsf{a}, m) & = \begin{cases} \langle \mathcal{D}, \Sigma', \Gamma \setminus Th(\mathcal{D} \cup \Sigma') \rangle & \text{if } \mathcal{D} \cup \Sigma \models pre(\mathsf{a}, m) \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathcal{M}(\mathbf{adopt}(\varphi), m) & = \begin{cases} \langle \mathcal{D}, \Sigma, \Gamma \cup \{\varphi\} \rangle & \text{if } \not\models \neg\varphi \text{ and } \Sigma \not\models \varphi \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathcal{M}(\mathbf{drop}(\varphi), m) & = \langle \Sigma, \Gamma \setminus \{\psi \in \Gamma \mid \psi \models \varphi\} \rangle
\end{array}$$

where $\Sigma' = \Sigma \oplus post(\mathsf{a}, m)$ and $Th(T) = \{\varphi \in \mathcal{L} \mid T \models \varphi\}$.

As discussed above, an action rule r is of the form **if** ψ **then** a. An action rule specifies that action a may be performed if the mental state condition ψ and the precondition of a hold. In that case, we say that action a is an *option*. At runtime, a GOAL agent non-deterministically selects an action from the set of options. This is expressed in the following transition rule, describing how an agent gets from one mental state to another.

Definition 4.5. *(Action Semantics)*
Let m be a mental state, and $r =$**if** ψ **then** a be an action rule. The transition relation $\xrightarrow{\mathsf{a}}$ is the smallest relation induced by the following transition rule.

$$\frac{m \models_c \psi \quad \mathcal{M}(\mathsf{a}, m) \text{ is defined}}{m \xrightarrow{\mathsf{a}} \mathcal{M}(\mathsf{a}, m)}$$

The execution of a GOAL agent results in a *computation*. We define a computation as a sequence of mental states and actions, such that each mental state can be obtained from the previous by applying the transition rule of Definition 4.5. As GOAL agents are non-deterministic, the semantics of a GOAL agent is defined as the *set* of possible computations of the GOAL agent, where all computations start in the initial mental state of the agent.

Definition 4.6. *(Computation)*
A *computation*, typically denoted by t, is an infinite sequence of mental states $m_0, \mathsf{a}_0, m_1, \mathsf{a}_1, m_2, \mathsf{a}_2, \ldots$ such that for each i we have that $m_i \xrightarrow{\mathsf{a}_i} m_{i+1}$ can be derived using the transition rule of Definition 4.5, or for all $j > i$, $m_j = m_i$ and $m_i \not\xrightarrow{\mathsf{a}} m'$ for any a and m'. The meaning S of a GOAL agent with initial mental state m_0 is the set of all computations starting in that state. We also write t_i^m to denote the ith mental state and t_i^{a} to denote the ith action.

Observe that a computation is infinite by definition, even if the agent is not able to perform any action anymore from some point in time on. Also note that the concept of an agent computation is a general notion in program semantics that is not particular to GOAL. The notion of a computation can be defined for any agent programming language that is provided with a well-defined operational semantics.

4.2.2.3 Verification Framework

A formal verification framework exists to verify properties of GOAL agents [3]. This verification framework allows for compositional verification of GOAL agents and has been related to Intention Logic [20]. The language GOAL thus is firmly rooted in agent theory.
The verification logic for GOAL is based on Linear Temporal Logic extended with modal operators for beliefs and goals. In addition the logic includes a set of Hoare rules to reason about actions [3]. The setup of the verification framework has some similarities with that for Unity [8]. To obtain a verification logic for GOAL agents temporal operators are added on top of mental state conditions to be able to express temporal properties over traces. Additionally an operator **start** is introduced to be able to pinpoint the start of a trace.[25]

Definition 4.7. *(Temporal Language: Syntax)*
The *temporal language* $\mathcal{L}_G$, with typical elements χ, χ', is defined by:

[25] Here, only the temporal semantics is presented. The compositional verification of an agent program also requires reasoning about actions. [3] introduces so-called Hoare rules to do so. In [20] an operator $[\mathsf{a}]\chi$ for reasoning about actions is introduced as this makes it easier to relate the verification logic for GOAL to Intention Logic [10].

$$\chi \in \mathcal{L}_G ::= \textbf{start} \mid \psi \in \mathcal{L}_m \mid \neg\chi \mid \chi \wedge \chi \mid \chi \mathcal{U} \chi$$

The semantics of $\mathcal{L}_G$ is defined relative to a trace t and time point i.

Definition 4.8. *(Temporal Language: Semantics)*
The truth conditions of sentences from $\mathcal{L}_G$ given a trace t and time point i are inductively defined by:

$$\begin{array}{ll}
t, i \models \textbf{start} & \text{iff } i = 0, \\
t, i \models \mathbf{B}\phi & \text{iff } t_i^m \models_c \mathbf{B}\phi, \\
t, i \models \mathbf{G}\phi & \text{iff } t_i^m \models_c \mathbf{G}\phi, \\
t, i \models \neg\varphi & \text{iff } t, i \not\models \varphi, \\
t, i \models \varphi \wedge \psi & \text{iff } t, i \models \varphi \text{ and } t, i \models \psi, \\
t, i \models \bigcirc\psi & \text{iff } t, i+1 \models \psi, \\
t, i \models \varphi \mathcal{U} \psi & \text{iff } \exists j \geq i : t, j \models \psi \text{ and } \forall i \leq k < j : t, k \models \varphi
\end{array}$$

Using the $\mathcal{U}$ operator, other temporal operators such as the "sometime in the future operator" $\Diamond$ and the "always in the future operator" $\Box$ can be defined by $\Diamond\psi ::= \mathtt{true}\mathcal{U}\psi$ and $\Box\psi ::= \neg\Diamond\neg\psi$.
The temporal logic introduced above has provided a basis for a Maude [9] implementation for the GOAL language which facilitates model checking of GOAL agents. Maude has been used to verify the Blocks World agent discussed in this chapter.

4.2.3 Software Engineering Issues

A key step in the development of a GOAL agent is the design of the domain knowledge, the concepts needed to represent the agent's environment in its beliefs and the goals of the agent. As it has been discussed above, GOAL does not commit to any particular knowledge representation language to represent the beliefs and goals of an agent. In section 4.2.2.1 we have abstracted away from any particular knowledge representation language and defined an abstract knowledge representation technology. This abstract knowledge representation has been defined such that it makes clear what the minimal requirements are that a particular knowledge representation language should satisfy in order to facilitate integration into GOAL. Although the current implementation has integrated Prolog as the technology for knowledge representation, in principle, other languages such as Answer Set Programming [1], expert system languages such as CLIPS [26], database languages such as SQL [7], or a language such as PDDL [17] also fit the definition of a knowledge representation technology in section 4.2.2.1 and could have been used as well.
The option to integrate other knowledge representation technologies than Prolog in an agent programming language may facilitate programmers as agent programming per se does not require a programmer to learn a new and specific knowl-

edge representation language but the programmer may choose its own favorite knowledge representation tool instead. In principle this flexibility also allows the integration of, for example, legacy databases. The GOAL interpreter provides an interface that facilitates such integration in Java.
The GOAL interpreter provides other interfaces that facilitate connecting GOAL to an environment or to middleware infrastructure on top of which GOAL agents are run. The interface to an environment is generic and abstracts from the implementation language used to run the environment. At the time of writing, as the GOAL interpreter has been written in Java, Java has been used to connect GOAL agents to an environment. Our view is that this interface can be used and allows the integration of GOAL agents into a larger application, part of which has been written in Java or other languages.

4.2.4 Other features of the language

In this section we briefly discuss some other features of the GOAL language that are important in order to write practical applications. As the main aim of this chapter is to introduce the core concepts that distinguish GOAL from other languages, we only discuss some of the issues that are involved in the development of GOAL agents.

Environments and Sensing

Agents with incomplete information that act in an environment which possibly inhabits other agents need to have sensors for at least two reasons. First, sensors provide an agent with the ability to acquire new information about its environment previously unknown to it and thus to *explore* its environment. Second, sensors provide an agent with the ability to acquire information about changes in its environment that are not caused by the agent itself and thus to *keep track* of the current state of its environment.
In GOAL, sensing is not represented as an explicit act of the agent but a *perceptual interface* is defined between the agent and the environment that specifies which percepts an agent will receive from the environment. A GOAL agent thus does not actively perform sense actions (except for the case where the environment makes such actions available to an agent). Each time after a GOAL agent has performed an action the agent processes any *percepts* it may have received through its perceptual interface. Percepts represent "raw data" received from the environment the agent is operating in. The percept interface is part of the environment interface to connect GOAL agents to an environment.

Multi-Agent Systems

GOAL facilitates the development and execution of multiple GOAL agents. A multi-agent GOAL system needs to be specified by means of a *mas file*. A mas file in GOAL is a recipe for running a multi-agent system. It specifies which agents should be launched when the multi-agent system is launched and which GOAL source files should be used to initialize those agents. GOAL allows for the possibility that multiple agents instantiate a single GOAL agent file. Various features are available to facilitate this. In a mas file one can associate multiple agent names with a single GOAL file. Each agent name additionally can be supplied with a list of optional arguments. These options include the number of instances of an agent, indicated by `#nr`, that should be launched. This option is available to facilitate the launching of large numbers of agents without also having to specify large numbers of different agent names. Other options allow to initialize an agent with a particular set of beliefs specified in a separate file using `#include:filename.bb`. The beliefs in the file `filename.bb` are simply *added* to the belief base specified in the agent file. This option allows for the launching of a multi-agent system with a set of agents that, for example, share the same domain knowledge but have different beliefs about the state of the environment. The `#override:filename.bb` option is provided to completely override and replace the initial beliefs specified in the GOAL agent file. The overriding of a by the `#override:filename.bb` option simply replaces all beliefs in the initial belief base specified in the GOAL file; this is implemented by using the file `filename.bb` to initialize the belief base of the agent instead of loading the beliefs specified in the GOAL file into the agent's belief base. Similar options are available for other sections such as the **goals** and **action-spec** sections of a GOAL agent.
GOAL does not support explicit constructs to enable the mobility of agents. The main concern in the design of the language is to provide appropriate constructs for programming rational agents whereas issues such as mobility are delegated to the middleware infrastructure layer on top of which GOAL agents are run.

Communication at the Knowledge Level

Communication in the current implementation of GOAL is based on a simple "mailbox semantics", very similar to the communication semantics of 2APL [12]. Messages received are stored in an agent's mailbox and may be inspected by the agent by means of queries on special, reserved predicates `sent`(*agent*,*msg*) and `received`(*agent*,*msg*) where *agent* denotes the agent the message has been sent to or received from, respectively, and *msg* denotes the content of the message expressed in a knowledge representation language.
Although a "mailbox semantics" can be used to write agents that communicate messages, such a semantics leaves too much to the programmer. We feel that a semantics is needed that facilitates programming with the high-level concepts used to write agents such as beliefs and goals. Agent communication at the knowledge

level should facilitate communication between agents about their beliefs and goals. At the time of writing, it seems that there is no commonly agreed approach to incorporate communication into agent programming languages. Various languages take different approaches. The "mailbox semantics" of *2APL* is based on communication primitives `Send(receiver,performative,content)` with the effect of adding `sent(Receiver, Performative, Content)` to the sender's mailbox and `received(Receiver, Performative, Content)` to the receiver's mailbox. A similar construct is available in *Jason* [4]. However, the effect of performing a `.send(Receiver,tell,Content)` where `tell` is a specific instance of a performative is that the receiving agent adds the `Content` of the received message to its belief base instead of the fact that a message has been received.
The semantics of communication in agent programming languages seems rather poor compared to more theoretical frameworks such as FIPA. FIPA introduces many primitive notions of agent communication called *speech acts*. The broad range of speech act types identified, however, may complicate writing agent programs and it makes more sense to us to restrict the set of communication primitives provided by an agent programming language. In this respect we favor the approach taken by *Jason* which limits the set of communication primitives to a core set. We would prefer a set of primitives that allows communication of declarative content only in line with our aim to provide an agent programming language that facilitates declarative programming. We believe this is still an evolving area that requires more research. It would be useful, from a more practical perspective, to gain more experience about what would be useful communication primitives that facilitate the programming of multi-agent systems.

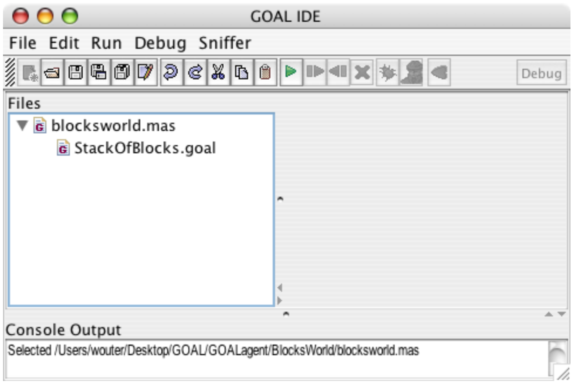
4.3 Platform

4.3.1 Available tools and documentation

The GOAL interpreter can be obtained by downloading the GOAL installer. For the most up to date version as well as information about the GOAL agent programming language the reader may visit

`http://mmi.tudelft.nl/~koen/goal.html`

Here also additional references to GOAL-related publications can be found. The language comes with an Integrated Development Environment (IDE) which allows editing and debugging of GOAL agents. The IDE is illustrated in Figures 4.3 and 4.4. Figure 4.3 shows the IDE after loading a mas file into the IDE. Upon loading a mas file, all files related to the same project are loaded and the plain text files (inlcuding GOAL files) are ready for editing. `jar` files related to environments cannot be edited.

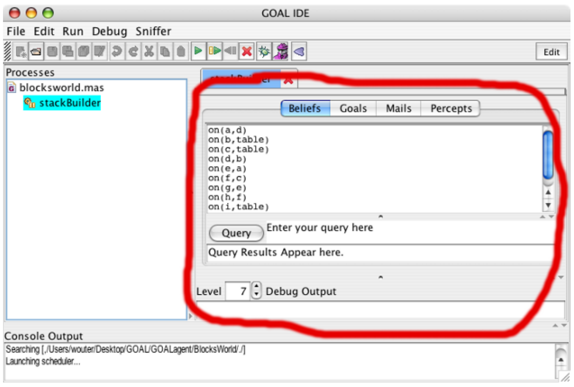


blocksworld.mas
StackOfBlocks.goal

```
Selected /Users/wouter/Desktop/GOAL/GOALagent/BlocksWorld/blocksworld.mas
```

Fig. 4.3 GOAL Integrated Development Environment

Alternatively, a loaded GOAL mas file can be executed from the IDE and the IDE is switched automatically to the run environment. Various options are available here to a user to monitor and debug GOAL agents. Figure 4.4 shows the introspector that is associated with each agent that is part of the multi-agent system that has been launched.



blocksworld.mas
stackBuilder

```
on(a,d)
on(b,table)
on(c,table)
on(d,b)
on(e,a)
on(f,c)
on(g,e)
on(h,f)
on(i,table)
```

```
Searching [./Users/wouter/Desktop/GOAL/GOALagent/BlocksWorld/./]
Launching scheduler...
```

Fig. 4.4 GOAL Agent Introspector

The introspector shows the agent's beliefs and goals, and any percepts and messages received. The knowledge, action rules and action specfications which are static parts of a GOAL agent are not shown here but may be inspected by inspecting the GOAL agent program text. The debugging functionality provided by the IDE can be used to trace the operation of an agent at various levels of granularity, e.g. at the inference level which allows tracing belief and goal inferences as well

as at higher levels which allows tracing of action selection only. Additionally, a sniffer is available to monitor message exchanges between agents.
GOAL comes with documentation discussing the language, IDE and some examples that are distributed with the language as well. A manual is provided for GOAL, including a discussion of the main language features, the IDE, installation and some advice on troubleshooting, and can be obtained from the site referenced above. The development of a tutorial is planned.

4.3.2 Standards compliance, interoperability and portability

The implementation of GOAL has been tested and runs on most well-known platforms. The system has been tested on Windows XP, Windows Vista 32bit, OSX Tiger, OSX Leopard (Intel only), and Linux with Ubuntu or Suse 10.1. The GOAL interpreter has been written in Java and needs SUN Java version 1.5 or higher. The design of the interpreter has been structured such that it provides a "plugin framework" that, in principle, can be instantiated with various knowledge representation technologies in line with the discussion in section 4.2.2.1 and various middleware systems that facilitate message passing and distributed computing on multiple machines. This has been achieved by defining a number of *interfaces* that specify what functionality the GOAL interpreter expects to be provided by the knowledge representation technologies or middleware systems. Similarly, an interface has been created that specifies how the GOAL interpreter can be connected to environments, e.g. a robot system or a simulated environment such as the Blocks World.
The requirements on the knowledge representation language used are minimal but the choice may introduce additional dependencies that may have consequences for portability. The current implementation integrates and uses SWI-Prolog [42] as the knowledge representation technology. As SWI-Prolog runs on most well-known operating systems, this does not introduce any severe restrictions, but other choices may do so. The use of SWI-Prolog does have implications for the number of agents that may be run on a single machine, however. Since by default SWI-Prolog reserves 100MB for any instance of a SWI-Prolog engine, in combination with the memory capacity of a machine on which the GOAL interpreter is run, this constrains the number of agents that may be run on that machine. Creating additional GOAL agents that go beyond this limit requires distributing these agents on multiple machines.
Similarly, the GOAL interpreter does not depend on any particular middleware infrastructure. The current implementation uses JADE [2] to facilitate interoperability with other systems that are built on top of JADE, but in principle any other middleware system that provides for message passing and the distributed execution of agents on multiple machines may be chosen. The middleware on top of which GOAL is run may also introduce additional dependencies or constraints

on the GOAL interpreter. We did not encounter any severe problems, however, while running GOAL on top of JADE on the platforms listed above.
The GOAL framework does not itself provide support for open systems nor for heterogeneous agents. GOAL agents are particular agents defined by their beliefs, goals and action rules that facilitate decision making. GOAL agents may nevertheless interact with other types of agents whenever these agents run on top of the same middleware infrastructure and exchange messages using the facilities provided by this infrastructure.

4.3.3 Other features of the platform

The current state of the GOAL platform is still a prototype. The core of the GOAL framework is stable and well-defined in several papers [3, 19, 21, 22] and has been implemented in the GOAL interpreter. GOAL will be distributed under the GPL open source license.The GOAL language is aimed at providing a general-purpose programming language for rational agents at the knowledge level. As it does not commit to any particular knowledge representation technology, domain or middleware infrastructure (see also section 4.3.2), users and/or developers of agent systems are provided with the tools to extend the GOAL interpreter with other knowledge representation technologies, and to implement other environments to run agents in.

4.4 Applications supported by the language and/or the platform

The GOAL agent programming language provides a high-level language for programming agents. The language provides high-level concepts such as beliefs, goals and action rules to select actions. GOAL is a general purpose agent programming language, but is most suitable for developing systems of rational agents that derive their choice of action from their beliefs and goals. It is not targeted at any specific application in particular, but may be most beneficially used in domains that are familiar from the traditional planning competitions. The Blocks World example discussed in this chapter provides an example of such a domain, but other domains such as the transportation domain may also provide good examples. GOAL agents provide additional flexibility and robustness as also illustrated by the Blocks World example. This is achieved by a flexible action selection mechanism based on action rules. The GOAL interpreter has been used in education to program agents that operate in a toy world and similarly a multi-agent system for cleaning dirt in a grid world has been written. We are currently looking at more serious applications among which a system of agents that negotiate by exchanging qualitative information besides the traditional bids in an alternating offer protocol and a Philips iCat robot with a cognitive control layer that interacts with humans while playing a game.

Acknowledgements I would like to thank everyone who has contributed to the development of GOAL, by helping to implement the language, developing the theoretical foundations, or contributing to extensions of GOAL. The list of people who have been involved one way or the other in the GOAL story so far, all of which I would like to thank are: Lăcrămioaria Aştefănoaei, Frank de Boer, Mehdi Dastani, Wiebe van der Hoek, Catholijn Jonker, John-Jules Ch. Meyer, Peter Novak, M. Birna van Riemsdijk, Tijmen Roberti, Nick Tinnemeier, Wouter de Vries. Special thanks go to Paul Harrenstein, who suggested the acronym GOAL to me, and Wouter Pasman, who did most of the programming of the GOAL interpreter. I would also like to thank Hilko Derde for useful remarks that helped to improve this chapter.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Pres (2003)
2. Bellifemine, F., Caire, G., Greenwood, D. (eds.): Developing Multi-Agent Systems with JADE. No. 15 in Agent Technology. John Wiley & Sons, Ltd. (2007)
3. de Boer, F., Hindriks, K., van der Hoek, W., Meyer, J.J.: A Verification Framework for Agent Programming with Declarative Goals. Journal of Applied Logic **5**(2), 277–302 (2007)
4. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
5. Bratman, M., Israel, D., Pollack, M.: Plans and resource-bounded practical reasoning. In: R. Cummins, J.L. Pollock (eds.) Philosophy and AI: Essays at the Interface, pp. 1–22. The MIT Press (1991)
6. Bratman, M.E.: Intentions, Plans, and Practical Reasoning. Harvard University Press (1987)
7. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. of KDE **1**(1) (1989)
8. Chandy, K.M., Misra, J.: Parallel Program Design. Addison-Wesley (1988)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285**(2), 187–243 (2002)
10. Cohen, P.R., Levesque, H.J.: Intention Is Choice with Commitment. Artificial Intelligence **42**, 213–261 (1990)
11. Cook, S., Liu, Y.: A Complete Axiomatization for Blocks World. Journal of Logic and Computation **13**(4), 581–594 (2002)
12. Dastani, M.: 2APL: a practical agent programming language. Journal Autonomous Agents and Multi-Agent Systems **16**(3), 214–248 (2008)
13. Dastani, M., Hindriks, K.V., Novak, P., Tinnemeier, N.A.: Combining multiple knowledge representation technologies into agent programming languages. In: Proceedings of the International Workshop on Declarative Agent Languages and Theories (DALT'08) (2008). To appear
14. Davidson, D.: Actions, reasons and causes. In: Essays on Actions and Events. Oxford University Press (1980)
15. Davis, R., Shrobe, H.E., Szolovits, P.: What is a knowledge representation? AI **14**(1), 17–33 (1993)
16. Dretske, F.: Explaining Behavior: Reasons in a World of Causes. The MIT Press (1995)
17. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)

18. Gupta, N., Nau, D.S.: On the Complexity of Blocks-World Planning. Artificial Intelligence **56**(2-3), 223–254 (1992)
19. Hindriks, K.: Modules as policy-based intentions: Modular agent programming in goal. In: Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'07), vol. 4908 (2008)
20. Hindriks, K., van der Hoek, W.: GOAL agents instantiate intention logic. In: Proceedings of the 11th European Conference on Logics in Artificial Intelligence (JELIA'08), pp. 232–244 (2008)
21. Hindriks, K., Jonker, C., Pasman, W.: Exploring heuristic action selection in agent programming. In: Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS'08) (2008)
22. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.C.: Agent Programming with Declarative Goals. In: Proceedings of the 7th International Workshop on Agent Theories Architectures and Languages, *LNCS*, vol. 1986, pp. 228–243 (2000)
23. Hindriks, K.V., van Riemsdijk, M.B.: Using temporal logic to integrate goals and qualitative preferences into agent programming. In: Proceedings of the International Workshop on Declarative Agent Languages and Theories, vol. 5397, pp. 215–232 (2008)
24. Hindriks, K.V., van Riemsdijk, M.B., van der Hoek, W.: Agent programming with temporally extended goals. In: Proceedings of the 8th International Conference on Autonomous Agents and Multi-Agent Systems (2009)
25. van der Hoek, W., van Linder, B., Meyer, J.J.: An Integrated Modal Approach to Rational Agents. In: M. Wooldridge (ed.) Foundations of Rational Agency, Applied Logic Series 14, pp. 133–168. Kluwer, Dordrecht (1999)
26. Jackson, P.: Introduction To Expert Systems, 3rd edn. Addison-Wesley (1999)
27. Lifschitz, V.: On the semantics of strips. In: M. Georgeff, A. Lansky (eds.) Reasoning about Actions and Plans, pp. 1–9. Morgan Kaufman (1986)
28. McCarthy, J.: Programs with common sense. In: Proceedings of the Teddington Conference on the Mechanization of Thought Processes, pp. 75–91. Her Majesty's Stationary Office, London (1959)
29. McCarthy, J.: Ascribing mental qualities to machines. Tech. rep., Stanford AI Lab, Stanford, CA (1979)
30. Meyer, J.J.C., van der Hoek, W.: Epistemic Logic for AI and Computer Science. Cambridge: Cambridge University Press (1995)
31. Newell, A.: The Knowledge Level. Artificial Intelligence **18**(1), 87–127 (1982)
32. Pearl, J.: Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference. Morgan Kaufmann (1988)
33. Plotkin, G.: A Structural Approach to Operational Semantics. Tech. Rep. DAIMI FN-19, University of Aarhus (1981)
34. Rao, A.S., Georgeff, M.P.: Intentions and Rational Commitment. Tech. Rep. 8, Australian Artificial Intelligence Institute (1993)
35. van Riemsdijk, M.B., Dastani, M., Meyer, J.J.C.: Goals in conflict: semantic foundations of goals in agent programming. Autonomous Agents and Multi-Agent Systems (2008). Online
36. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall (2003)
37. Schoppers, M.: Universal plans for reactive robots in unpredictable environments. In: Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI'87) (1987)
38. Scowen, R.S.: Extended BNF - A generic base standard. `http://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf` (1996)
39. Shoham, Y.: Agent-oriented programming. Artificial Intelligence **60**, 51–92 (1993)
40. Slaney, J., Thiébaux, S.: Blocks World revisited. Artificial Intelligence **125**, 119–153 (2001)
41. Sterling, L., Shapiro, E.: The Art of Prolog, 2nd edn. MIT Press (1994)
42. `http://www.swi-prolog.org/` (2008)