Künstl Intell (2012) 26:61–67
DOI 10.1007/s13218-011-0155-2

PROJEKT

PLATAS—Integrating Planning and the Action Language Golog

Jens Claßen · Gabriele Röger · Gerhard Lakemeyer · Bernhard Nebel

Published online: 26 November 2011

Abstract Action programming languages like Golog allow to define complex behaviors for agents on the basis of action representations in terms of expressive (first-order) logical formalisms, making them suitable for realistic scenarios of agents with only partial world knowledge. Often these scenarios include sub-tasks that require sequential planning. While in principle it is possible to express and execute such planning sub-tasks directly in Golog, the system can performance-wise not compete with state-of-the-art planners. In this paper, we report on our efforts to integrate efficient planning and expressive action programming in the PLATAS project. The theoretical foundation is laid by a mapping between the planning language PDDL and the Situation Calculus, which is underlying Golog, together with a study of how these formalisms relate in terms of expressivity. The practical benefit is demonstrated by an evaluation of embedding a PDDL planner into Golog, showing a drastic increase in performance while retaining the full expressiveness of Golog.

J. Claßen (✉) · G. Lakemeyer
Knowledge-Based Systems Group, RWTH Aachen University, Ahornstraße 55, 52056 Aachen, Germany
e-mail: classen@kbsg.rwth-aachen.de

G. Lakemeyer
e-mail: gerhard@cs.rwth-aachen.de

G. Röger · B. Nebel
Foundations of Artificial Intelligence, University of Freiburg, Georges-Köhler-Allee 52, 79110 Freiburg, Germany

G. Röger
e-mail: roeger@informatik.uni-freiburg.de

B. Nebel
e-mail: nebel@informatik.uni-freiburg.de

1 Introduction

Action programming languages like Golog [14] allow to define complex behaviors for agents on top of their high-level actions, which are expressed in some form of expressive (first-order) logical formalism such as the Situation Calculus [17, 24]. Among other things, the logical representation allows to model realistic scenarios of agents with only incomplete information about the world state, where further information has then to be gathered at runtime by sensing.

For defining an agent's behavior, one can either use *programming*, meaning the programmer completely predetermines the agent's actions, or one can resort to *planning*, i.e. provide a description of a desired goal and leave it to the system to find a suitable course of action. An advantage of Golog is that it gives the programmer the freedom to choose anywhere between these two extremes by freely combining deterministic and nondeterministic parts.

As an example, consider a mobile robot in an office environment. One of its tasks is to deliver letters between mailboxes in the different offices, and the procedure to take all letters out of a mailbox m may look like this:

proc *takeAllLetters*(m)
  while$(\exists l : \mathit{letter}.\ \mathit{In}(l, m))$ **do**
    $(\pi\ l : \mathit{letter})\ [\mathit{In}(l, m)?; \texttt{take\_out}(l, m)]$.

It reads as follows: As long as there are objects of type *letter* in m, select (π) some *letter* l such that $\mathit{In}(l, m)$ holds and take it out. Note that in addition to constructs known from imperative programming languages like sequential composition (";") and **while** loops, Golog programs may contain *nondeterministic* constructs such as the pick operator π. The program is viewed as a plan *sketch* where certain parts have been left open, and it is left to the system to fill these gaps.

After taking the letters, the robot should deliver them to their recipients. This is a typical case where planning is needed: We know the task requires a number of `move` actions (to travel between locations), interspersed with `put_in` actions (to put letters in mailboxes), but narrowing it further down in terms of a Golog program is difficult. One option is the following, highly nondeterministic subprogram:

while $(\neg \mathit{Goal})$ **do** (πa) $[\phi(a)?; a]$.

The program says that as long as the *Goal* formula is currently not true, the agent has to nondeterministically choose some action a for which ϕ applies and execute it. In our example, *Goal* would express that letters for which the recipient is known are delivered to their destination mailboxes:

$$\forall l : \mathit{letter}.\ (\exists m : \mathit{mailbox}\ \mathit{Addressee}(l, m)) \supset \mathit{Delivered}(l)$$

ϕ is some criterion to filter suitable actions; in the example it may express that a is among the actions `put_in` and `move`:

$$\exists l_1, l_2 : \mathit{loc}\ (a = \texttt{move}(l_1, l_2)) \vee \\ \exists l : \mathit{letter}, m : \mathit{mailbox}\ (a = \texttt{put\_in}(l, m)).$$

Any successful execution of the above program corresponds to an action sequence that reaches a situation where *Goal* holds, i.e. a plan. When the Golog interpreter does a lookahead to search for such an action sequence, it thus performs nothing else than classical, sequential planning.

Planning is hence both necessary and possible in principle, but the performance of Golog in these cases is generally poor compared to state-of-the-art planners [2, 4]. The reason is that current Golog implementations typically resort to blind search for lookahead, while the focus of planning research ever since the introduction of STRIPS [8] has been on designing systems of high efficiency, so that a wide range of techniques and heuristics is available today. It seems natural to ask how this progress can be exploited to improve the performance of Golog. It is conceivable to do this by simply copying techniques from successful planners into Golog. However, this endeavor is undesirable not only as it would mean a pure re-implementation of existing results, but also because it will have to be reiterated for future developments in the area. Instead, it would be much more desirable to plug in any existing planner by means of a simple interface.

Since there is already a common interface to state-of-the-art planning systems, namely the *Planning Domain Description Language* (PDDL) [19], we use the following approach: Whenever the Golog interpreter encounters a planning sub-problem, we translate it into PDDL, call a PDDL planner on it, and continue the execution with the returned plan. Note that classical planning alone is not sufficient for the runtime control of the agent, as there are additional important tasks. Most notably, the agent typically has only partial knowledge about its environment and has to gather necessary information at runtime using its sensors. For example, before executing *takeAllLetters*(m), the robot has to use the sensing action `look_into`(m) to learn which letters there are in m. Golog then takes care of updating the knowledge base of the agent with the results of the sensing actions.

For the embedding, two issues arise. First, we have to ensure that the translation between Golog and PDDL is correct in the sense that the same plans are admitted. Moreover, there is the question how the two formalisms relate in terms of expressivity, i.e. which subset of the source formalism is representable by the target formalism. In this paper, we report on how we addressed these issues within the PLATAS project (*Planning Techniques and Action Languages*), which is part of the DFG-funded project cluster *LogWiss*.[1]

2 The Situation Calculus and Golog

The *Situation Calculus* [17, 24] is a dialect of first-order logic for reasoning about dynamic domains. Changes in the world are assumed to be the result of *primitive actions*, which are performed by some implicit agent and modeled by terms like $\texttt{move}(l_1, l_2)$. Properties which are affected by performing such actions are called fluents and can be predicates like $\mathit{Holding}(l, s)$ or functions like $\mathit{position}(\mathit{robot}, s)$. The last argument of a fluent is a *situation*, which should be understood as the current history of actions that have been executed. The constant S_0 is used to denote the initial situation, and when a is an action and s a situation, then $do(a, s)$ denotes the situation that results from performing a in s. For example $\mathit{Holding}(l, do(\texttt{take\_out}(l, m), S_0))$ means that the agent is holding letter l after taking it out of m. A particular domain is described by a *basic action theory* (BAT), which is a set of Situation Calculus formulas consisting of:

1. Sentences that describes the initial situation, e.g.

   $$\neg\exists x\ \mathit{Holding}(x, S_0) \wedge \forall l : \mathit{letter}\, \neg \mathit{Delivered}(l, S_0),$$

2. a precondition formula for each action, e.g.[2]

   $$\mathit{Poss}(\texttt{take\_out}(x, y), s) \equiv \\ \exists l : \mathit{loc}.\ \mathit{At}(\mathit{robot}, l, s) \wedge \mathit{At}(y, l, s) \wedge \mathit{In}(x, y, s);$$

3. for each fluent, a successor state axiom (SSA) encoding how the predicate is affected by actions, e.g.

   $$\mathit{Holding}(x, do(a, s)) \equiv \exists y(a = \texttt{take\_out}(x, y)) \vee \\ \mathit{Holding}(x, s) \wedge \neg\exists y(a = \texttt{put\_in}(x, y)).$$

α	primitive action
$\phi?$	test
$\delta_1; \delta_2$	sequence
$\delta_1 \mid \delta_2$	nondeterministic choice
$\pi x.\delta(x)$	nondeterministic choice of argument
δ^*	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2	conditional
while ϕ **do** δ	loop
proc $P(\vec{x})$ $\delta(\vec{x})$	procedures

Fig. 1 Basic set of Golog constructs

Based on the Situation Calculus, *Golog* allows the definition of complex actions called *programs*. The standard variant supports the constructs shown in Fig. 1. An important aspect is that apart from imperative constructs such as conditionals, loops and procedures, a program can also contain *nondeterministic* parts, e.g. $\delta_1 \mid \delta_2$ (do either δ_1 or δ_2) and δ^* (perform δ zero or more times). A program thus represents an incomplete sketch of a problem solution, where nondeterministic parts constitute gaps to be filled by the system.

Variants of Golog extend this basic set of constructs by additional functionality: ConGolog [5] adds concurrency in form of interleaved processes, exogenous events and interrupts. A problem for real-world applications though is that programs are executed off-line in ConGolog, meaning that the interpreter first analyzes the entire program to find a full conforming action sequence before the first action is actually executed. Moreover, especially scenarios where the agent has incomplete knowledge about its environment make it necessary to gather information at runtime through sensing. IndiGolog [6] is an extension that tackles both issues and is hence particularly suited for our purposes. Programs are generally executed on-line without lookahead, which is only applied to subprograms explicitly marked via the search operator $\Sigma(\delta)$. Furthermore it supports sensing actions.

3 PDDL

PDDL [18, 19] was introduced in 1998 as the input language for the First International Planning Competition (IPC) [18] and has, after several revisions, by now become a defacto standard for the formulation of planning domains and tasks.

In contrast to the Situation Calculus where changes to the world are described fluent-centric by SSAs, PDDL is an action-centric language where actions are specified by their applicability conditions and their effect on the world. For example, the `move` action from our running example could be formulated in PDDL as shown in Fig. 2. A full PDDL task is specified by the *domain* description that describes the dynamics of the world (mostly by such actions) and a *task* description that specifies task-specific aspects, e.g. additional constants, the initial state and the goal. Note that in contrast to BATs there is always a single, fully specified initial state. Since actions are deterministic, PDDL planning systems have always full information about the world state.

```
(:action move
 :parameters
   (?l1 - loc ?l2 - loc)
 :precondition
   (and (at robot ?l1) (connected ?l1 ?l2))
 :effect
   (and (at robot ?l2)
      (not (at robot ?l1))
      (forall (?x - obj)
        (when (holding ?x)
           (and (at ?x ?l2)
             (not (at ?x ?l1)))))))
```

Fig. 2 PDDL definition of action `move`

The development of PDDL has been strongly connected to the IPC and, therefore, has been oriented on the capabilities of state-of-the-art planning systems. At the same time, the introduction of new language features gave impulses to new developments. To name a few, PDDL now also covers numeric and temporal aspects (like simple and continuous durative actions that can be executed concurrently [9]), and preferences [10]. Since most of these features are tackled in special competition tracks, planning systems do not have to support all features but can specialize on certain fragments.

4 Mapping PDDL to the Situation Calculus

In order to establish a common semantical basis for PDDL and Golog, we show how a PDDL problem description is mapped to a corresponding BAT. The embedding of planners into Golog actually requires a translation in the other direction, which we obtain by taking the back-image of our mapping. This has two advantages: First, it is simpler since the Situation Calculus is more expressive than PDDL. Second, the mapping can be viewed as an alternative, declarative semantics for PDDL whose semantical definition [9] extends the state-transitional one for STRIPS [15] to the additional features of PDDL. In this definition, states are represented as sets of literals and a transition is specified by the addition and deletion of literals. Lin and Reiter [16] identified this as problematic in particular for incomplete theories and showed that there is an exact correspondence between STRIPS and certain forms of BATs in the sense that adding

[1] www.computational-logic.org/content/projects/wisslogabc.php

[2] Free variables are understood as ∀-quantified from the outside.

and deleting literals can be viewed as a mere *mechanism* for computing the *progression* of such a BAT, which means updating it to reflect the changes due to an action. We generalize this result to incrementally larger subsets of PDDL.

The mapping requires to switch from the action-centric view of PDDL to the fluent-centric one of the Situation Calculus. Pednault already sketched the general idea when he first introduced ADL [21], and Reiter [24] formalized it in his famous solution to the frame problem: For each fluent F, we collect all positive effects on it in a single positive effect axiom, and all negative ones in a single negative effect axiom. If `move` is the only action that affects `at`, we get:

$$\gamma^+_{At} = \exists l : loc.(a = \texttt{move}(l, y)) \\ \wedge (x = robot \vee Holding(x, s)),$$

$$\gamma^-_{At} = \exists l : loc.(a = \texttt{move}(y, l)) \\ \wedge (x = robot \vee Holding(x, s)).$$

Then we assume that these effects indeed describe *all and only* possible changes to the fluent, yielding the SSA:

$$At(x, y, do(a, s)) \equiv \gamma^+_{At} \vee At(x, y, s) \wedge \neg\gamma^-_{At}.$$

In words, At is true after executing a if it was made true (γ^+_{At}) or if it was already true and not made false (γ^-_{At}). We verified that a similar construction is correct for the ADL fragment of PDDL, which extends basic STRIPS with conditional effects as well as negated, disjunctive, and quantified preconditions, but does not add higher language features like durative actions or numeric functions: In this fragment, an update to a PDDL state w.r.t. some action corresponds to the progression of the constructed BAT through that action. The progression is guaranteed to exist due to the closed-world and domain-closure assumptions being made in PDDL [2].

We then extended these results to a fragment that includes the temporal and numeric features of PDDL [3]. It uses an explicit notion of time, where a plan has to specify at which time points actions happen. In addition to simple instantaneous actions, it includes durative actions whose execution stretches over some time *interval*. In our robot scenario, for instance, it is reasonable to model `move` as durative since the robot needs a certain amount of time to get from one location to another. Our treatment of time and durative actions in the Situation Calculus is based on work by Pinto [22] and Reiter [23]: actions are extended with an additional argument for the execution time, e.g. $\texttt{take\_out}(x, y)$ becomes $\texttt{take\_out}(x, y, t)$. Furthermore, a durative action a' is encoded by splitting it into two simple actions $start(a', t)$ and $end(a', t)$ that mark the start and end points of its execution interval. We then have a special fluent keep track of which durative actions are in progress, using the SSA:

$$Performing(a', do(a, s)) \equiv \exists t.\ a = start(a', t) \vee \\ Performing(a', s) \wedge \neg\exists t'.\ a = end(a', t').$$

In PDDL, preconditions and effects of durative actions are annotated with one of the time qualifiers "`at start`", "`at end`", and "`overall`", where the former two refer to the start and end *time points*, respectively, and the latter to the *open interval* between them. As long as a precondition or an effect is only concerned with the start (end) time point, our mapping to the Situation Calculus is straightforward: We simply treat it as a precondition or effect of the corresponding $start(a', t)$ ($end(a', t)$) action. However, there might also be inter-temporal effects. In the example of a durative variant of our robot's `move` action, we may have the effect that *at the end* an object arrives together with the robot at the destination if the object is held *at the beginning*:

```
(when (at start (holding ?x))
      (at end   (at ?x ?l2))))
```

In such cases, we introduce auxiliary fluents that "memorize" whether the condition was true at the beginning:

$$Mem^{start}_{move}(x, l_1, l_2, do(a, s)) \equiv \\ \exists t'.\ a = start(\texttt{move}(l_1, l_2, t')) \wedge Holding(x, s) \vee \\ Mem^{start}_{move}(x, l_1, l_2, do(a, s)) \wedge \\ \neg\exists t.\ a = end(\texttt{move}(l_1, l_2, t)).$$

We also came up with representations of the further temporal features of PDDL such as continuous effects, invariants, and timed initial literals. We proved correctness in the sense that valid temporal PDDL plans correspond to sequences of *start* and *end* actions admitted by the BAT. The remaining features of full PDDL such as trajectory constraints, preferences and derived predicates can be mapped similarly [11].

5 Expressivity

Planners are typically designed for specific PDDL fragments since restricting the scope of the system allows for more specialized and efficient algorithms. We are therefore interested in using a fragment that is as small as possible when we formulate a Golog sub-task in PDDL. To understand which fragment of BATs corresponds to which fragment of PDDL, we compare their expressivity using the compilation scheme framework introduced by Nebel [20]. Compilation schemes compare how concisely planning domains and plans can be expressed in different formalisms, not how concisely an individual planning instance can be expressed. The intuition is that it is justifiable to perform significant work on translating a domain description from one formalism to another, as long as this remains a one-off effort, and individual *instances* of the domain can subsequently be transformed efficiently.

As we are measuring the *expressivity* of a formalism, the mapping may use arbitrary computational resources, but we require that the result is at most polynomially sized and that the transformation of the domain description does not depend on the initial state and goal. In contrast, the translation of the initial state and the goal is very limited: it should be efficiently computable and not depend on the whole specification. If such a mapping is solution preserving, i.e. there is a plan for the original task iff there is a plan for the result task, we call it a compilation scheme. To compare the expressivity of two planning formalisms we moreover have to measure the size of generated plans. If it is possible to formulate a compilation scheme so that the size of the shortest plan for the result task is bounded linearly by the size of an optimal plan for the source task, we conclude that the target formalism is at least as expressive as the source formalism. If plans are required to grow faster, the source formalism is more expressive. In our project, it turned out that compilation schemes always directly provided a polynomial-time translation to PDDL and the resulting plan can directly be interpreted as a solution situation for the source task.

A particularly interesting subset of PDDL is its ADL fragment, which is more expressive than the rather restrictive STRIPS but still supported by a large number of planning systems. Based on earlier work [7], we identified a maximal subset of the Situation Calculus that is equally expressive [25, 26]. Put shortly, we can compile all BATs that provide full information on the initial state in an explicit (non-compact) form into the ADL fragment. More precisely, this Situation Calculus fragment can briefly be characterized as all BATs whose initial database consists of exactly the following sentences (where c_i denotes a constant):

1. For each relational fluent F there is an expression

$$F(x_1, \ldots, x_n, S_0) \equiv$$
$$(x_1 = c_{11} \wedge \cdots \wedge x_n = c_{1n})$$
$$\vee \cdots \vee (x_1 = c_{m1} \wedge \cdots \wedge x_n = c_{mn}).$$

2. There are analogous expressions for all situation-independent predicates.
3. There are analogous expressions for all functions except constants and action functions with object arguments.
4. There are unique names axioms $c_i \neq c_j$ for each pair c_i, c_j of different constants.
5. Optionally, there may be a domain closure axiom of the form $(x = c_1) \vee \cdots \vee (x = c_n)$.

6 Experimental Results

Based on our results we extended IndiGolog with access to a planning system that supports the ADL fragment, and conducted experiments to assess the practical benefit of the embedding. A simple simulator played the role of the agent's environment by generating appropriate sensing results.

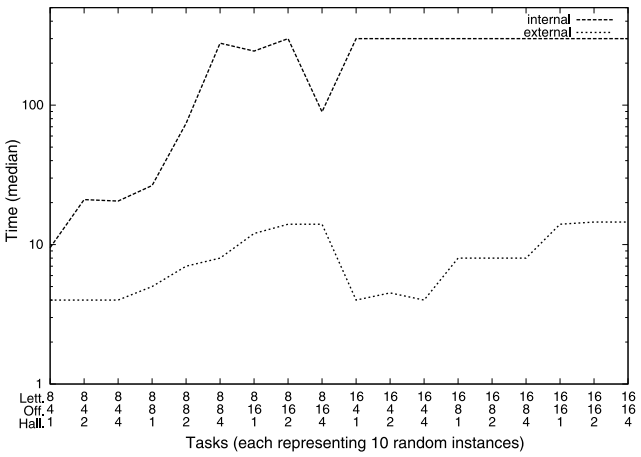

Fig. 3 Median runtimes in seconds

In one experiment, we tested the mail robot domain of our running example. We let the building consist of a number of locations, each of which is either an office or a hallway. Each office is connected to some hallway, and hallways are connected to one another. Furthermore, each mailbox serves both for incoming and outgoing mail and is located in some office. Initially, each letter is in one of the mailboxes. The robot executes the following program: *As long as not all letters are delivered* (which we signal from outside), *randomly pick a mailbox, go to it, get all letters from it and deliver them.* In our encoding, moving to the mailbox is one planning sub-task, getting the letters out is achieved via the procedure *takeAllLetters* from the introduction, and delivering the letters is another planning subproblem.

We randomly generated benchmark scenarios for different numbers of offices (= number of mailboxes), hallways and letters, with 10 instances of each combination. We then compared the performance of two variants of our system which differ in how planning sub-tasks are treated: In one, the internal iterative deepening mechanism of Golog is used to solve them. In the other one, each planning task is translated into a PDDL problem as described above, after which an external PDDL planner is called, and execution is continued in Golog with the returned solution plan. The planner we used is the state-of-the-art Fast Downward planning system [12] in a configuration that guarantees optimal plans (A* search with the lmCut heuristic [13]). All experiments were conducted on a 64 bit Intel Core2 Duo PC with 2.66 Ghz and 2 GiB of main memory and a 300 seconds timeout per task.

With the external PDDL planner, IndiGolog could complete all 180 benchmark tasks within the time limit, while it

only did so for 63 with its internal search, not solving any of the 16 letter instances. Figure 3 shows the median runtimes for both variants. The difference between the two is considerable (note the logarithmic scale), showing that the embedding definitely improves performance and scalability.

7 Conclusion

The areas of planning on the one hand and action languages on the other hand have developed largely independently for many years. The planning community was mostly concerned with developing efficient systems, whereas the focus in action logics was more on formalisms of high expressivity. The PLATAS project is a contribution to integrate both fields, yielding results of both theoretical and practical relevance: The mapping presented in Sect. 4 can be viewed as a declarative semantics for PDDL, which facilitates the analysis of the language through expressing its properties in terms of entailments in a well-understood logic. The comparison of PDDL with the Situation Calculus by means of compilation schemes as described in Sect. 5 furthermore gives us insight into how the two formalisms relate in terms of mutual expressivity, and hence which subset of one formalism can be translated into the other. Both aspects form the theoretical basis for the practical integration of efficient planners into a Golog system. As shown in Sect. 6, this approach enables a drastic increase in performance, while it retains the full expressivity of the underlying action language.

Future work will for one extend the above beyond PDDL to other planning formalisms. For another, we aim to increase the efficiency of Golog also on sub-tasks where the agent has only partial knowledge, possibly at the cost of losing the completeness of the underlying reasoning method. A first step in this direction has already been explored [1].

Acknowledgements PLATAS is part of the LogWiss research cluster. It is funded by the Deutsche Forschungsgemeinschaft (DFG) under grants La 747/13-2, La 747/14-1, Ne 623/10-1, and Ne 623/10-2.

References

1. Claßen J, Lakemeyer G (2009) Tractable first-order Golog with disjunctive knowledge bases. In: Proc Commonsense. UTSePress, Broadway, pp 27–33
2. Claßen J, Eyerich P, Lakemeyer G, Nebel B (2007) Towards an integration of Golog and planning. In: Proc IJCAI 2007. AAAI Press, Menlo Park, pp 1846–1851
3. Claßen J, Hu Y, Lakemeyer G (2007) A situation-calculus semantics for an expressive fragment of PDDL. In: Proc AAAI 2007. AAAI Press, Menlo Park, pp 956–961
4. Claßen J, Engelmann V, Lakemeyer G, Röger G (2008) Integrating Golog and planning: An empirical evaluation. In: Proc NMR 2008, pp 10–18
5. De Giacomo G, Lespérance Y, Levesque HJ (2000) ConGolog, a concurrent programming language based on the situation calculus. Artif Intell 121(1–2):109–169
6. De Giacomo G, Levesque HJ, Sardiña S (2001) Incremental execution of guarded theories. ACM Trans Comput Log 2(4):495–525
7. Eyerich P, Nebel B, Lakemeyer G, Claßen J (2006) Golog and PDDL: What is the relative expressiveness. In: Proc PCAR 2006. University of Western Australia Press, Nedlands, pp 93–104
8. Fikes R, Nilsson NJ (1971) STRIPS: A new approach to the application of theorem proving to problem solving. Artif Intell 2(3/4):189–208
9. Fox M, Long D (2003) PDDL2.1: An extension to PDDL for expressing temporal planning domains. J Artif Intell Res 20:61–124
10. Gerevini A, Long D (2005) Plan constraints and preferences in PDDL3. Tech Rep RT 2005-08-47, Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia
11. Han J (2009) A declarative semantics of a subset of PDDL with constraints and preferences. Diploma thesis, Department of Computer Science, RWTH Aachen University, Aachen, Germany
12. Helmert M (2006) The Fast Downward planning system. J Artif Intell Res 26:191–246
13. Helmert M, Domshlak C (2009) Landmarks, critical paths and abstractions: What's the difference anyway? In: Proc ICAPS 2009, pp 162–169
14. Levesque HJ, Reiter R, Lespérance Y, Lin F, Scherl RB (1997) GOLOG: A logic programming language for dynamic domains. J Log Program 31(1–3):59–83
15. Lifschitz V (1987) On the semantics of STRIPS. In: Reasoning about actions and plans: proceedings of the 1986 workshop. Morgan Kaufmann, San Mateo, pp 1–9
16. Lin F, Reiter R (1997) How to progress a database. Artif Intell 92(1–2):131–167
17. McCarthy J, Hayes P (1969) Some philosophical problems from the standpoint of artificial intelligence. In: Machine intelligence, vol 4. American Elsevier, New York, pp 463–502
18. McDermott DV (2000) The 1998 AI planning systems competition. AI Mag 21(2):35–55
19. McDermott D, Ghallab M, Howe A, Knoblock C, Ram A, Veloso M, Weld D, Wilkins D (1998) PDDL—The planning domain definition language—Version 1.2. Tech Rep CVC TR-98-003, Yale Center for Computational Vision and Control
20. Nebel B (2000) On the compilability and expressive power of propositional planning formalisms. J Artif Intell Res 12:271–315
21. Pednault EPD (1989) ADL: Exploring the middle ground between STRIPS and the situation calculus. In: Proc KR 1989. Morgan Kaufmann, San Mateo, pp 324–332
22. Pinto J (1994) Temporal reasoning in the situation calculus. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada
23. Reiter R (1998) Sequential, temporal GOLOG. In: Proc KR 1998. Morgan Kaufmann, San Mateo, pp 547–556
24. Reiter R (2001) Knowledge in action: logical foundations for specifying and implementing dynamical systems. MIT Press, Cambridge
25. Röger G, Nebel B (2007) Expressiveness of ADL and Golog: Functions make a difference. In: Proc AAAI 2007. AAAI Press, Menlo Park, pp 1051–1056
26. Röger G, Helmert M, Nebel B (2008) On the relative expressiveness of ADL and Golog: The last piece in the puzzle. In: Proc KR 2008. AAAI Press, Menlo Park, pp 544–550

Jens Claßen received his diploma degree in computer science in 2005 from RWTH Aachen University. Since then he has been working as a research assistant at the Knowledge-Based Systems Group in Aachen. His research interests include planning, reasoning about actions and change and epistemic logics.

Gabriele Röger received her diploma degree in computer science from the University of Freiburg in 2006. Since then she has been working as a research assistant in the research group on *Foundations of Artificial Intelligence* at the University of Freiburg. She was awarded as co-author with the 2008 AAAI outstanding paper award. Her research interests include planning and combinatorial search.

Gerhard Lakemeyer received his Ph.D. from the University of Toronto in 1990 and currently heads the Knowledge-Based Systems Group at RWTH Aachen University. His research interests include knowledge representation and cognitive robotics. He has published more than one hundred scientific papers and has served on numerous program committees. He is an ECCAI Fellow and a member of the Editorial Board of Artificial Intelligence and other journals.

Bernhard Nebel is full professor of Computer Science at the University of Freiburg and chair of the research group *Foundations of Artificial Intelligence*. He was elected as an ECCAI fellow in 2001, as an AAAI fellow in 2010 and is member of the German Academy of Science *Leopoldina*. His research interests include knowledge representation and reasoning, planning, and robotics.