Why the Monkey Needs the Box: A Serious Look at a Toy Domain

Selim T. Erdoğan, Paolo Ferraris, Vladimir Lifschitz, Wanwan Ren
University of Texas at Austin
Department of Computer Sciences
{selim,otto,vl,rww6}@cs.utexas.edu

Abstract

"Toy worlds" involving actions, such as the Blocks World and the Monkey and Bananas domain, are often used by researchers in the areas of commonsense reasoning and planning to illustrate and test their ideas. Many of the axioms found in descriptions of these toy worlds are expressions of general-purpose knowledge, though they are often cast in a form only useful for solving one specific problem and are not faithful representations of general facts that can be used in other domains. Instead of using such domain-specific axioms for each problem, we are building a general-purpose library of action descriptions which can be referred to in descriptions of many action domains. The library is being written in the modular action description language MAD, an extension of the action language $\mathcal{C}+$. In this paper we present an initial version of some of our library modules, along with a new formalization of the Monkey and Bananas domain that uses the library. Most of the axioms in this formalization come from the library, with only a few domain-specific axioms needed.

1 Introduction

"Toy worlds" involving actions are often used by researchers in the areas of commonsense reasoning and planning to illustrate and test their ideas. Many of the axioms found in descriptions of these toy worlds are expressions of general-purpose knowledge, though they are often cast in a form only useful for solving one specific problem and are not faithful representations of general facts that can be used in other domains.

For instance, familiar formalizations of the Monkey and Bananas domain [Green, 1969, Section III], [Giunchiglia *et al.*, 2004, Section 4.3] postulate that to perform the action of grasping the bananas, the monkey has to be on the box and, at the same time, under the bananas. These preconditions can be viewed as special cases of a general principle:

> Normally, to perform an action that affects an object *x*, an agent has to be at the same place as *x*. (1)

Being on the box and under the bananas is what guarantees that this precondition is satisfied when *x* is a bunch of bananas hanging from the ceiling.

Approaching the bananas in the horizontal plane can be accomplished by performing a single action—walking. Approaching the bananas in the vertical direction is more of a challenge, because the monkey can't float in the air. More generally:

> Normally, it is impossible for an object not to be supported by anything. (2)

For instance, the monkey is initially supported by the floor; after climbing the box, he is supported by the box. The box is always supported by the floor. If we adopt the general understanding of "supported" as "held in the current position" then we can also say that the bananas are supported by the ceiling in the initial state, and by the monkey in the final state.

General principles (1) and (2) explain why the monkey needs the box to get the bananas. Principle (1), applied to the box as *x*, also explains why, before pushing the box to the place under the bananas, the monkey needs to walk towards the box.

We would like to design a general-purpose database, or library, of commonsense facts related to actions, such as (1) and (2), which can be referred to in formal descriptions of action domains. In this paper we present an initial, rudimentary version of the library (Section 2), along with a new formalization of the Monkey and Bananas domain that uses the library (Section 3). Most of the axioms in this formalization come from the library, with only a few domain-specific axioms needed. The work on testing the library that we have done so far is described in Section 4.

The database is being written in the modular action description language MAD [Lifschitz and Ren, 2006], with several additional constructs that we found useful. MAD is an extension of the action language $\mathcal{C}+$, described in [Giunchiglia *et al.*, 2004]. The main distinctive feature of this extension is the possibility of referring to other action descriptions in the definition of a new domain [Erdoğan and Lifschitz, 2006], which is crucial for our purposes. A MAD action description is a list of "modules" $M_1, \dots, M_n$. A module M_i can use, or "import," any of the modules $M_1, \dots, M_{i-1}$, possibly in several ways. Each module describes a set of interrelated fluents and actions. Import statements allow the user to characterize new fluents and actions by relating them to others, introduced earlier. When a module M_i imports a module M_j, it "inherits" the knowledge encoded in M_j, possibly restricted to a specialized context and expressed in different notation.

The idea of making a general-purpose database of commonsense knowledge is not new. John McCarthy [1987] pointed out that AI programs suffer from a lack of generality and cited this as the key problem in AI. The CYC project [Lenat and Guha, 1990; Matuszek *et al.*, 2006], under development for over 20 years, is the best known attempt to create a very large database of common sense. The KM Component Library [Barker *et al.*, 2001] is a collection of reusable general purpose knowledge components. The knowledge base that we are building differs from these projects mainly in two ways. First, we focus on properties of actions, rather than commonsense knowledge in general. Reasoning about actions and causation is an important special case of commonsense reasoning, and in recent years research in this area has led to significant advances. Second, we represent knowledge in an action description language in the sense of [Gelfond and Lifschitz, 1998]. The semantics of such languages is based on a simple, well-understood concept of a transition system. The close relationship between action languages and logic programming [Lifschitz and Turner, 1999] will hopefully allow us to solve reasoning and planning problems involving the knowledge base that we are building using methods of answer set programming [Lifschitz, 2002].

2 A General-Purpose Knowledge Base

The formalization of the "locality principle" (1) in Section 2.5 below relies on three ideas. First, actions may be executed by "agents." Second, actions may be associated with "themes"—things that are affected by them. Third, things have "locations." These ideas are formalized in the first three modules of the knowledge base. Then we introduce several actions that change the locations of objects. The next group of modules deals with the "support principle" (2) and with actions that affect the support relation.

2.1 Actors and Themes

```
module ACTOR;
  sorts
    Agent;
  constants
    Actor(Agent,action): Boolean;
  variables
    u: Agent;
    a: action;
  axioms
    default -Actor(u,a);
endmodule;
```

Comment: The module above talks about objects of two sorts, agents and actions. The first sort is declared at the beginning of the module; the second sort is predefined in (our extended version of) the language MAD. `Actor` is a relation between an agent and an action. The only axiom in this module tells us that this relation is subject to the closed world assumption: by default, it is presumed not to hold.

Digression: This paper does not review details of the syntax and semantics of MAD, but we will say the following about the `default` construct. Its meaning in MAD is the same as in $\mathcal{C}+$: it can be characterized in terms of the nonmonotonic causal logic described in [Giunchiglia *et al.*, 2004], which is closely related to the system of default logic from [Reiter, 1980]. Similarly, the precise meaning of the MAD construct `inertial`, used in module `PLACE` below, is closely related to Reiter's solution to the frame problem in terms of a normal default.

The next module is similar:

```
module THEME;
  sorts
    Thing;
  constants
    Theme(Thing,action): Boolean;
  variables
    x: Thing;
    a: action;
  axioms
    default -Theme(x,a);
endmodule;
```

2.2 Places and Movement

```
module PLACE;
  sorts
    Thing; Place;
  constants
    Location(Thing): fluent(Place);
  variables
    x: Thing;
  axioms
    inertial Location(x);
endmodule;
```

Comment: The location of a thing is a place-valued fluent. (Thus locations may change with time.) Locations satisfy the commonsense law of inertia: by default, they are presumed not to change.

```
module MOVE;
  import PLACE;
  constants
    Move(Thing,Place): action;
  variables
    x: Thing;
    p: Place;
  import THEME;
  axioms
    exogenous Move(x,p);
    Move(x,p) causes Location(x)=p;
    Theme(x,Move(x,p));
endmodule;
```

Comment: Moving a thing `x` to a place `p` is an action. This action is exogenous (it can be executed or not executed at will), and it causes the location of `x` to be `p`. Furthermore, `x` is a theme of this action. Note that the constants `Location` and Theme, used in the axioms, are not explicitly declared here. According to the semantics of MAD, the effect of the two import statements in this module is essentially to include in it all declarations and axioms of modules PLACE and THEME. (However, the variables declared in PLACE and THEME are considered "local" to those modules and cannot be used in MOVE unless they are explicitly declared.)

```
module GO;
  sorts
    Agent; Thing; Place;
  inclusions
    Agent << Thing;
  constants
    Go(Agent, Place): action;
  variables
    u: Agent;
    x: Thing;
    p: Place;
  import MOVE;
    Move(x,p) is exists u (u=x &
                              Go(u,p));
  import ACTOR;
  axioms
    Actor(u,Go(u,p));
endmodule;
```

Comment: This module tells us that

(i) the set of agents is a subset of the set of things,

(ii) the action Go has the same properties as Move except that its first argument is an agent, rather than an arbitrary thing, and

(iii) this action is executed by the agent that moves.

Technically, (ii) is expressed using the statement

```
  import MOVE;
    Move(x,p) is exists u (u=x &
                              Go(u,p));
```

(see [Lifschitz and Ren, 2006] on the semantics of statements of this form). The formula to the right of is describes the action that consists in executing Go(x,p) if x is an agent, and is not executable otherwise. The statement asserts that this action has the properties postulated for Move(x,p) in module MOVE.

```
module CARRY;
  sorts
    Agent; Thing; Place;
  constants
    Carry(Agent,Thing,Place): action;
  variables
    u: Agent;
    x: Thing;
    p: Place;
  import MOVE;
    Move(x,p) is Carry(u,x,p);
  import GO;
    Go(u,p) is Carry(u,x,p);
  axioms
    Actor(u,Carry(u,x,p));
    Theme(x,Carry(u,x,p));
endmodule;
```

Comment: Executing the action Carry(u,x,p) involves both moving and going, so that it affects the locations of both x and u. This action is executed by u, and its theme is x.

2.3 Support

```
module SUPPORT;
  sorts
    Thing; Supporter;
  inclusions
    Thing << Supporter;
  constants
    Support(Thing): fluent(Supporter);
    Supported(Thing,Supporter): sdFluent;
  variables
    x, y: Thing;
    s:    Supporter;
  axioms
    inertial Support(x);
    Supported(x,s) if Support(x)=s;
    Supported(x,s) if Support(x)=y &
                      Supported(y,s);
    default -Supported(x,s);
    constraint -Supported(x,x);
endmodule;
```

Comment: A thing is supported either by another thing or by a "supporter" of a different kind, such as the surface of the earth (in the next module, such supporters will be called "foundations"). This idea is expressed here by declaring the support of a thing to be a supporter-valued fluent. The first axiom makes supports subject to the commonsense law of inertia. The next three axioms define Supported as the transitive closure of the relation Support(x)=s.[1] This definition is needed to state the last axiom, which says that a thing cannot support itself even indirectly.

```
module FOUNDATION;
  import SUPPORT;
  sorts
    Foundation;
  inclusions
    Foundation << Supporter;
  variables
    x: Thing;
    f: Foundation;
  axioms
    constraint exists f Supported(x,f);
endmodule;
```

Comment: The axiom above formalizes the "support principle" (2): a thing is always supported by a foundation, possibly indirectly. It is not entirely satisfactory, because it does not allow exceptions, such as objects in free fall; nothing in this axiom corresponds to the word "normally" in (2). We

[1] Technically, Supported is a "statically determined fluent" (sdFluent); see [Giunchiglia *et al.*, 2004, Section 4.2].

would like to correct this eventually, using the nonmonotonic features of MAD.

The next three modules introduce actions that affect supports: `Mount` and its special cases `Climb` and `Grasp`.

```
module MOUNT;
  import SUPPORT;
  constants
    Mount(Thing,Supporter): action;
  variables
    x: Thing;
    s: Supporter;
  import THEME;
  axioms
    exogenous Mount(x,s);
    Mount(x,s) causes Support(x)=s;
    Theme(x,Mount(x,s));
endmodule;
```

```
module CLIMB;
  import ACTOR;
  sorts
    Thing; Supporter;
  inclusions
    Agent << Thing << Supporter;
  constants
    Climb(Agent,Supporter): action;
  variables
    u: Agent;
    x: Thing;
    s: Supporter;
  import MOUNT;
    Mount(x,s) is exists u (u=x &
                            Climb(u,s));
  axioms
    Actor(u,Climb(u,s));
    Theme(x,Climb(u,x));
endmodule;
```

Comment: According to the last axiom, the supporter `s` is a theme of `Climb(u,s)` if `s` is a thing. The declaration of `Theme` does not allow an object that is not a thing to be a theme of any action.

```
module GRASP;
  import ACTOR;
  sorts
    Thing; Supporter;
  inclusions
    Agent << Thing << Supporter;
  constants
    Grasp(Agent,Thing): action;
  variables
    u: Agent;
    x: Thing;
    s: Supporter;
  import MOUNT;
    Mount(x,s) is exists u (u=s &
                            Grasp(u,x));
  axioms
    Actor(u,Grasp(u,x));
    Theme(x,Grasp(u,x));
endmodule;
```

2.4 Relationship between Locations and Supports

```
module TOP;
  import PLACE;
  import SUPPORT;
  constants
    Top(Thing): sdFluent(Place);
  variables
    x, y: Thing;
    p:    Place;
  axioms
    Location(x)=p if Support(x)=y &
                     Top(y)=p;
endmodule;
```

Comment: If a thing `x` is supported by a thing `y` (and not by a foundation) then the location of `x` is determined by the location of `y`. In particular, moving `y` to another place will usually cause `x` to move to another place too. To express this idea, we introduced here the fluent `Top(y)`, which represents the place where all things supported by `y` are located. This module merely provides a mechanism for describing how the location of a thing is determined by the location of its support, but specific assumptions about what the top is need to be specified whenever the module is imported.

```
module AT;
  import PLACE;
  import FOUNDATION;
  constants
    At(Foundation,Place): Boolean;
  variables
    x: Thing;
    f: Foundation;
    p: Place;
  axioms
    default -At(f,p);
    Location(x)=p if Support(x)=f &
                     At(f,p);
endmodule;
```

Comment: The declaration of `Location` allows us to talk about the location of a thing, but not about the location of a foundation. It may happen, however, that a foundation `f` is associated with a place `p`, in the sense that all things supported by `f` are at `p`. For instance, all things supported by a horizontal surface are at the same level. This idea can be expresses using the Boolean fluent `At(f,p)`.

2.5 General Properties of Actions

```
module NOCONCURRENCY;
  variables
    a, a1: explicitAction;
  axioms
    nonexecutable a & a1 if a!=a1;
endmodule;
```

Comment: By importing this module we can express the idea that actions cannot be executed concurrently. The use of the predefined sort `explicitAction`, rather than `action`, indicates that, in the process of grounding, both `a` and `a1` are to be replaced by action names which have been given "explicitly" by the user, and not by action names that are internally generated by the system upon encountering an `is` clause of an `import` statement. For instance, the module `MBF` below imports the library module `GO` to say that `Walk(P1)` is synonymous with `Go(Monkey,P1)`; `Walk` is an action name given explicitly in the module, though `Go` is redefined and will turn into an internally generated action name. Since only one of them is an explicitly named action, the axiom in module `NOCONCURRENCY` allows `Walk(P1)` and `Go(Monkey,P1)` to be true at the same time.

Finally, here is our formalization of the "locality principle" (1):

```
module LOCAL;
  import ACTOR;
  import THEME;
  inclusions
    Agent << Thing;
  import PLACE;
  variables
    u: Agent;
    x: Thing;
    a: action;
  axioms
    nonexecutable a
      if Actor(u,a) & Theme(x,a) &
         Location(u)!=Location(x);
endmodule;
```

Comment: This formalization is not entirely satisfactory, because it does not allow exceptions, such as the action of sending an e-mail, which affects a computer at another location. See the comment after module `FOUNDATION` in Section 2.3.

3 Monkey and Bananas

There is a monkey in a room, and a bunch of bananas hanging from the ceiling beyond his reach. There is also a box. The monkey can walk to the box, push it under the bananas and climb on it to reach the bananas.

We describe this domain in three steps. Module `MBF` describes what goes on at the floor level of the room, where the monkey can walk or push the box. Module `MBS`, which imports `MBF`, describes how the things in the domain are supported. Finally, module `MB`, which imports `MBS`, is a full description of the domain.

```
module MBF;
  import LOCAL;
  objects
    Monkey:     Agent;
    Box:        Thing;
    P1, P2, P3: Place;
  constants
    Walk(Place), PushBox(Place): action;
  variables
    u: Agent;
    x: Thing;
    p: Place;
  import GO;
    Go(u,p) is u=Monkey & Walk(p);


  import CARRY;
    Carry(u,x,p) is u=Monkey & x=Box &
                    PushBox(p);
  import NOCONCURRENCY;
endmodule;
```

Comment: There are no domain-specific axioms in this module. The properties of `Walk` are described by a reference to the library module `GO`, and the properties of `PushBox` by a reference to the library module `CARRY`.

```
module MBS;
  import MBF;
  import FOUNDATION;
  objects
    Bananas:         Thing;
    Floor, Ceiling: Foundation;
  import TOP;
  constants
    ClimbOn, ClimbOff, GetBananas: action;
  variables
    u: Agent;
    x: Thing;
    p: Place;
    s: Supporter;
  import GRASP;
    Grasp(u,x) is u=Monkey & x=Bananas &
                  GetBananas;
  import CLIMB;
    Climb(u,s) is u=Monkey &
                  ((s=Box & ClimbOn) |
                   (s=Floor & ClimbOff));
  axioms
    constraint Support(Box)=Floor;
    constraint Support(Monkey)=Floor |
               Support(Monkey)=Box;
    nonexecutable ClimbOff
               if Support(Monkey)=Floor;
    Top(x)=p if Location(x)=p;
endmodule;
```

Comment: We did not include the axiom

```
    nonexecutable ClimbOn
               if Support(Monkey)=Box;
```

similar to the nonexecutability axiom for `ClimbOff`, because it would have been redundant: the locality principle for levels ("vertical locations") introduced in the next module prevents the execution of the action `Climb(Monkey,Box)` whenever the monkey (agent) and the box (theme) are at different levels. In contrast, action `ClimbOff` is characterized by `Climb(Monkey,Floor)`, which doesn't have a theme since `Floor` is not a thing. The last axiom of this module tells us that things are straight: whatever is supported by `x` is at the same place as `x`. This assumption guarantees that after climbing on the box that is placed under the bananas the monkey will be under the bananas too.

```
module MB;
  import MBS;


  sorts
    Level;
  objects
    Lo, Hi: Level;
  constants
    Elevation(Thing): fluent(Level);
    TopLevel(Thing): sdFluent(Level);
  variables
    x: Thing;
    l: Level;
  import LOCAL;
    Place is Level;
    Location(x)=l is Elevation(x)=l;
  import TOP;
    Place is Level;
    Location(x)=l is Elevation(x)=l;
    Top(x)=l is TopLevel(x)=l;
  import AT;
    Place is Level;
    Location(x)=l is Elevation(x)=l;
  axioms
    default TopLevel(x)=l
          if Elevation(x)=l;
    TopLevel(Box)=Hi;
    At(Floor,Lo);
    At(Ceiling,Hi);
endmodule
```

Comment: The import of TOP in this module allows us to talk about "vertical locations" of things, that is to say, about their elevations. According to the first axiom, normally things are not tall: whatever is supported by x is at the same level as x. However, the box is an exception. The ceiling is high above the floor.

According to the semantics of MAD, the list consisting of the 15 library modules and the three domain-specific modules shown above decribes a transition system—a directed graph whose vertices correspond to possible states of the system, and edges correspond to the transitions between states that can be caused by execution of actions, or simpy by the passage of time—such trivial transitions are self-loops.

4 Testing

We mentioned earlier that MAD is an extension of the action language $\mathcal{C}+$. The semantics of MAD is defined by translating a modular action description into a $\mathcal{C}+$ description. We have implemented this process of translating a MAD description into a $\mathcal{C}+$ description, though the implementation of the reasoning engine is not yet complete. However, we were able to use the Causal Calculator[2] (CCALC), an implementation of a subset of $\mathcal{C}+$, to test our formalization of the Monkey and Bananas domain using the library modules.

[2] http://www.cs.utexas.edu/~tag/ccalc/ .

In order to test our formalization we gave CCALC queries and checked that its answers matched our expectations. For all of these queries, there were 3 locations, P1, P2, P3, as was declared in the module MBF. Here are some example queries. (In the queries below, all letters are in the lower case, to conform with the syntax of CCALC.)

1. The monkey is at p1*, the bananas are hanging from the ceiling over location* p2 *and the box is at* p3*. We would like to find the shortest possible plan (no longer than 10 steps) in which the monkey gets the bananas.*

This planning problem can be described as follows:

```
:- query
maxstep :: 1..10;
0: location(monkey)=p1,
   location(bananas)=p2,
   support(bananas)=ceiling,
   location(box)=p3;
maxstep: support(bananas)=monkey.
```

Given this query, CCALC first reports that there is no solution within 3 steps, and gives a solution with four steps, consisting of executing actions walk(p3), pushbox(p2), climbon and getbananas one by one.

2. Are there any other plans with four steps? What about plans with exactly five steps?

The CCALC queries for these questions are very similar to the previous query. As expected, there are no other plans with four steps, though there are 16 plans with exactly five steps. Of these, 5 involve waiting (i.e. performing no actions during a step). The others involve frivolous actions (e.g. starting off by taking a walk to p3 before walking to the box at p2—maybe the monkey just wanted to get a better look at the bananas before pushing the box!) or trivial actions (e.g. walking to p1 when the monkey is already at p1).

Besides making plans, we may also give queries testing individual aspects of our library modules:

3. The monkey is at p1*, and the box is someplace other than* p1*. Is it possible to push the box?*

We represent this as follows: (++ is disjunction in CCALC)

```
:- query
maxstep :: 1;
0: location(monkey)=p1,
   location(box)\=p1,
   pushbox(p1) ++ pushbox(p2) ++
                  pushbox(p3).
```

CCALC reports that no solutions can be found embodying such a scenario. The monkey cannot push the box because this would be a violation of the "locality principle" (1), expressed in module LOCAL.

4. The monkey is not supported by the box. Where may things (the monkey, the box, the bananas) be and what may they be supported by?

```
:- query
maxstep :: 0;
0: support(monkey)\=box.
```

CCALC reports that there are 72 configurations where the monkey is not supported by the box. The box and the monkey are always supported by the floor, due to the domain-specific constraints in module MBS. On the other hand the bananas may be supported by the floor, the ceiling, the monkey and the box. When the bananas are on the floor, there are 27 configurations corresponding to the three locations each thing may be at. Likewise when the bananas are hanging from the ceiling. If, however, the bananas are supported by the monkey or the box, then there are only 9 configurations in each case, since the location of the bananas is determined by the location of its supporter. This principle comes from the the library module TOP.

It was not possible to feed the automatically generated $\mathcal{C}+$ description directly into CCALC, because CCALC implements the "definite" fragment of $\mathcal{C}+$ whereas the $\mathcal{C}+$ description auto-generated from the MAD description contains "nondefinite" axioms. (See [Giunchiglia *et al.*, 2004, Sec. 5] for more about the definite fragment of $\mathcal{C}+$.) Every time an action or a fluent is characterized in terms of others, in an `import` statement, a nondefinite axiom is introduced. In order to use CCALC, it was necessary to take the generated $\mathcal{C}+$ description and manually turn the nondefinite $\mathcal{C}+$ axioms into equivalent definite axioms, following the methods described in [Erdoğan and Lifschitz, 2006].

In the future, this transformation will be automated. We are also considering an alternative approach to query answering for nondefinite $\mathcal{C}+$ action descriptions that is based on the reduction of causal logic to answer set programming proposed in [Ferraris, 2007].

Another change in the automatically generated $\mathcal{C}+$ description that we had to do manually is related to the predefined sort `explicitAction`, used in the module NOCONCURRENCY, which has no counterpart in the input language of CCALC.

5 Conclusion

We presented an initial version of our library of general-purpose action descriptions and showed how the Monkey and Bananas domain may be formalized using this library. The new formalization contains three new modules with only 8 axioms, whereas the original $\mathcal{C}+$ formalization of the same domain, given in [Giunchiglia *et al.*, 2004] contains 31 axioms. This difference is due to the fact that most of the knowledge about the actions in the domain (`Walk`, `PushBox`, `ClimbOn`, `ClimbOff`, `GetBananas`) is part of the general-purpose library.

As we built up the library modules shown in this paper, we identified new language features that will be useful. We are currently working on an extension of the semantics of MAD that covers these additional features and on implementing this extended dialect of MAD.

The next step will be to extend the work described in this note to other action domains familiar from the literature on commonsense reasoning and planning.

Acknowledgements

We are grateful to Michael Gelfond, Yuliya Lierler and Ashu Manohar for useful discussions related to the topic of this paper. This research was partially supported by the National Science Foundation under Grant IIS-0412907.

References

[Barker *et al.*, 2001] Ken Barker, Bruce Porter, and Peter Clark. A library of generic concepts for composing knowledge bases. In *First International Conference on Knowledge Capture*, pages 14–21, 2001.

[Erdoğan and Lifschitz, 2006] Selim T. Erdoğan and Vladimir Lifschitz. Actions as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 377–387, 2006.

[Ferraris, 2007] Paolo Ferraris. A logic program characterization of causal theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2007. To appear.

[Gelfond and Lifschitz, 1998] Michael Gelfond and Vladimir Lifschitz. Action languages.[3] *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998.

[Giunchiglia *et al.*, 2004] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

[Green, 1969] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 219–240, 1969.

[Lenat and Guha, 1990] Douglas Lenat and R. V. Guha. *Buliding large knowledge-based systems*. Addison-Wesley, 1990.

[Lifschitz and Ren, 2006] Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 853–859, 2006.

[Lifschitz and Turner, 1999] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 92–106, 1999.

[Lifschitz, 2002] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[Matuszek *et al.*, 2006] Cynthia Matuszek, John Cabral, Michael Witbrock, and DeOliveira John. An introduction to the syntax and content of Cyc. In *Working Notes of the AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, 2006.

[3] `http://www.ep.liu.se/ea/cis/1998/016/` .

[McCarthy, 1987] John McCarthy. Generality in Artificial Intelligence. *Communications of ACM*, 30(12):1030–1035, 1987. Reproduced in [McCarthy, 1990].

[McCarthy, 1990] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy*. Ablex, Norwood, NJ, 1990.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.