Briefcases, Cars, and Airplanes

Selim T. Erdoğan and Vladimir Lifschitz
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-0233 USA
{selim,vl}@cs.utexas.edu

Abstract

Our goal is to develop a methodology for describing commonsense action domains on the basis of a library of actions of a general nature, such as moving an object to a new place. The use of this library in descriptions of specific domains will be similar to the use of libraries of standard subroutines in programming languages. We show how this idea can be applied to several domains familiar from the literature on commonsense reasoning and planning. These domains involve moving containers and vehicles—briefcases, cars and trucks, boats and airplanes. The action descriptions are written in (an extension of) the modular action description language MAD proposed by Lifschitz and Ren, which is now implemented on top of the Causal Calculator.

1 Introduction

This is the third in the series of preliminary reports on the work motivated by the idea of interaction between two directions of research in knowledge representation: the design of action description languages [Gelfond and Lifschitz, 1998] and the development of libraries of reusable, general-purpose knowledge components [Barker *et al.*, 2001]. In the first of these reports [Erdoğan and Lifschitz, 2006], we conjectured that a library of standard descriptions for a number of "basic" actions can facilitate writing, understanding and modifying action descriptions, and illustrated this idea by showing how the action *PushBox* in the Monkey and Bananas domain can be described as a special case of the "library action" *Move*.

The second report in this series [Lifschitz and Ren, 2006] defined a language called MAD (for *Modular Action Descriptions*) that will be used in this project. MAD is an extension of the language $\mathcal{C}+$ proposed in [Giunchiglia *et al.*, 2004]. Its main distinctive feature is the "import" construct, which allows the user to refer to action descriptions introduced earlier in the definition of a new domain.

In this paper, our approach is applied to several action domains familiar from the literature on commonsense reasoning and planning. Each of these domains has to do with containers that can change their locations along with their contents, or vehicles that move around along with their passengers and luggage. One of these examples is the briefcase that Ed Pednault used twenty years ago to carry a book to his office [Pednault, 1988]. Long before that, a boat was used by missionaries and cannibals to cross the river [Amarel, 1968], and a car driven by John McCarthy took him to the airport [McCarthy, 1959]. Years later, trucks and airplanes took packages to their destinations in the logistics domain [Veloso, 1992]. Examples of this kind are formalized here in an extended version of MAD in terms of general-purpose actions, such as *Move*, *Load* and *Unload*. "Library descriptions" of these actions are written in MAD also.

One advantage of this description style is that it is more natural, in the sense that humans often describe actions to each other in this way—by relating them to other actions. As observed in [Erdoğan and Lifschitz, 2006], the dictionary says that pushing is *moving by steady pressure*; this phrase explains the meaning of the word *push* not by listing the effects of this action, but by presenting it as a special case of another action, moving. In this paper we show also that this approach makes many action descriptions more concise, and that it makes it easier for us to recognize structural similarities between action domains.

We implemented the MAD language on top of the Causal Calculator (CCALC),[1] so that reasoning and planning problems for action domains described in MAD can now be solved automatically. All of the formalizations shown in this paper were tested using this implementation.

A closely related paper by Michael Gelfond [2006], entitled *Going places—notes on a modular development of knowledge about travel*, is directed towards "the development and implementation of a library of knowledge modules needed for axiomatization of journey—a movement of a group of objects from one place (the origin) to another (the destination)." It emphasizes the possibility of unexpected stops in the middle of a journey. Adding modular structure to the logic programming language CR-Prolog [Balduccini, 2007] in that paper is similar to adding modular structure to $\mathcal{C}+$ in [Lifschitz and Ren, 2006].

The applicability of the object-oriented paradigm to modeling dynamic domains is investigated by Joakim Gustafsson and Jonas Kvarnström [2004]. Their system is based on Temporal Action Logic [Doherty and Kvarnström, 2008].

[1] `http://www.cs.utexas.edu/users/tag/ccalc/` .

2 The MAD language and the briefcase domain

MAD is a member of the family of action languages [Gelfond and Lifschitz, 1998]—languages that serve to describe the effects of actions on states of the world. The semantics of action languages is defined in terms of "transition systems."

2.1 The briefcase domain as a transition system

Consider two sets of symbols, called *fluent constants* and *action constants*, along with a nonempty finite set of symbols assigned to each fluent constant c, called the *domain* of c. In this paper, a *transition system* is a directed graph whose vertices, or *states*, are functions that map every fluent constant to an element of its domain, and whose edges may be labeled by action constants. Intuitively, an edge from a state s to a state s' labeled a indicates that executing action a in state s can take the system to state s'; the absence of a label on an edge indicates that the transition can happen without executing any action.[2] In this framework, a planning problem can be thought of as the problem of finding a path in a transition system from a given initial state to a given goal state.

Action languages are formal languages for describing transition systems. We will illustrate the syntax of MAD using the following example from [Pednault, 1988]:

> Suppose that we have a world that consists of three objects—a briefcase, a dictionary, and a paycheck—each of which may be situated in one of two locations: the home or the office. Actions are available for putting objects in the briefcase, and for taking objects out, as well as for carrying the briefcase between the two locations. Initially, the briefcase, the dictionary, and the paycheck are at home; the paycheck is in the briefcase, but the dictionary is not. The goal is to have the briefcase and dictionary at the office and the paycheck at home.

We may represent this domain using five fluent constants. Three of them describe the locations of the briefcase, the dictionary, and the paycheck; the possible locations are the home and the office. The other two indicate whether the dictionary or the paycheck are in the briefcase. Out of the 32 combinations of values of these fluents, only 18 represent possible states of the world, because when the paycheck is in the briefcase, both have to be at the same place, and similarly for the dictionary.

Thus the transition system representing Pednault's domain has 18 states. Every edge of the system can be labeled by one of six action constants:

`PutIn(Paycheck)`, `TakeOut(Paycheck)`, `PutIn(Dictionary)`, `TakeOut(Dictionary)`, `MoveB(Home)`, `MoveB(Office)`.

The transition system has 60 edges; out of these, 18 are trivial (self-loops without a label), 9 are labeled by `MoveB(Home)` (carry the briefcase home, along with its contents), and 9 are labeled by `MoveB(Office)` (carry the briefcase to the office). The remaining 24 edges are labeled by `PutIn` and `TakeOut` actions.

[2] The MAD language, as defined in [Lifschitz and Ren, 2006], allows an edge to be labeled by a *set* of actions that are concurrently executed, possibly empty. In this paper, concurrent execution of actions is not allowed.

```
sorts
  Item;

inclusions
  Item << Thing;

module BRIEFCASE;

  objects
    Paycheck, Dictionary : Item;
    Briefcase            : Carrier;
    Home, Office         : Place;

  constants
    PutIn(Item),
    TakeOut(Item),
    MoveB(Place)         : action;

  variables
    i : Item;
    p : Place;

  import CARRIER;
    Load(i,Briefcase) is PutIn(i);
    Unload(i) is TakeOut(i);
    Move(Briefcase,p) is MoveB(p);
```

Figure 1: Formalization of the briefcase domain

2.2 The briefcase domain in MAD

A description of the briefcase domain in MAD is shown in Figure 1. This description contains several references to the library of basic action descriptions that we are building, and these references are explained as we encounter them. The full text of the current version of the library and its ontology are available at `http://www.cs.utexas.edu/users/tag/mad/library/`.

The description declares `Item` to be a sort, and it postulates that `Item` is a subsort of the sort `Thing`. The latter is "standard" in the sense that it is declared in the library ontology, along with its subsort `Carrier` and the sort `Place` that are used in Figure 1 also.

The module `BRIEFCASE` consists of four parts. The first three, beginning with the keywords `objects`, `constants`, and `variables`, consist of declarations. The fourth part imports the library module `CARRIER`; it is discussed in Section 2.3 below.

From the declarations we learn that `Paycheck`, `Dictionary`, `Briefcase`, `Home`, and `Office` are objects, and what their sorts are. Any of the words `PutIn` and `TakeOut`, followed by an object of sort `Item` in parentheses, is an action, as well as the word `MoveB` followed by an object of sort `Place`. Finally, `i` is a variable for objects

of sort `Item`, and `p` is a variable for objects of sort `Place`.

2.3 Importing the module CARRIER

The importing facility of MAD allows us to make use of existing action description modules when creating new action descriptions. In this way it is similar to the use of libraries of standard subroutines in programming languages.

The use of concepts of abstract algebra in the definition of a specific number system may be a better analogy. When we describe the set **R** of real numbers as a group relative to addition, with the neutral element 0, we say essentially that the axioms for groups

$$\begin{array}{ll} \forall x, y, z \in G & x \star (y \star z) = (x \star y) \star z, \\ \forall x \in G & x \star e = x, \\ \forall x \in G \, \exists y \in G & x \star y = e \end{array}$$

hold if

$$\begin{array}{l} G \text{ is } \mathbf{R}, \\ \star \text{ is } +, \\ e \text{ is } 0. \end{array}$$

Similarly, the import statement from Figure 1

```
import CARRIER;
  Load(i,Briefcase) is PutIn(i);
  Unload(i) is TakeOut(i);
  Move(Briefcase,p) is MoveB(p);
```

tells us that the action `PutIn(i)` has all properties that are postulated for the action `Load(x,c)` in the library module `CARRIER` when the thing `x` is an item and the carrier `c` is `Briefcase`, and similarly for the actions `TakeOut(i)` and `MoveB(p)`. For example, with this import the axiom

```
Move(x,p) causes Location(x)=p;
```

from module `CARRIER`[3] (where `x` is a variable for things) has the same effect as if we had written the axiom

```
MoveB(p) causes Location(Briefcase)=p;
```

in module `BRIEFCASE`.

The constant `Location`, used above, is declared in the library module `CARRIER` as follows:

```
Location(Thing): fluent(Place);
```

That is to say, the `Location` of a `Thing` is a (simple[4]) fluent, and its domain is the set of objects of sort `Place`. By importing `CARRIER` we get access to this fluent and to the other constants declared in that module.

One other assumption about the fluent `Location` in the module `CARRIER` is that it satisfies the commonsense law of inertia—the location of a thing is presumed to remain unchanged in the absence of information to the contrary. Furthermore, it is impossible to move a thing to its current location. These assumptions, just as the assumption about the effect of `Move(x,p)` on `Location(x)`, are "inherited" by `BRIEFCASE` from `CARRIER`. In the absence of a library of standard action descriptions, many such axioms would have to be explicitly included in module `BRIEFCASE`.

[3]To be precise, this axiom is found in the library module `MOVE`, which is imported by `CARRIER`.

[4]See [Giunchiglia *et al.*, 2004, Sections 4.2, 4.4] on the difference between simple fluents and statically determined fluents.

The fact that a carrier `c` is holding a thing `x` is described in module `CARRIER` by the truth-valued fluent `Holds(c,x)`.[5] Executing action `Load(x,c)` makes this fluent true, and executing `Unload(x)` makes it false.

According to the axioms of `CARRIER`, the action `Load(x,c)` is nonexecutable if `Location(x)` is different from `Location(c)`. For instance, the action of putting the dictionary in the briefcase cannot be executed when the dictionary is at home and the briefcase is at the office.

The effect of `Move(x,p)` on `Location(x)` in the axiom above is a direct effect of this action. Moving, loading and unloading can also have indirect effects. For instance, axioms of the module `CARRIER` show that moving a carrier affects the location of the objects that are held by it.

The precise semantics of `import` statements in MAD is defined in [Lifschitz and Ren, 2006].

2.4 Implementation and testing

We have an implementation of MAD that allows us to perform various kinds of reasoning (such as planning, prediction, postdiction) about action descriptions written in MAD. The implementation makes use of the Causal Calculator (CCALC) which is a system that can reason with the "definite" fragment of language $\mathcal{C}+$.

The system takes as input the library of basic action descriptions along with a domain-specific action description. It first turns this set of modules into an equivalent single-module description by eliminating import statements and incorporating their contents, with appropriate modifications, into the importing module, according to the semantics given in [Lifschitz and Ren, 2006]. A module without import sections is essentially a $\mathcal{C}+$ description. However, this description generally contains "nondefinite" axioms that CCALC cannot handle. Therefore we need to apply a further transformation, based on the methods outlined in [Erdoğan and Lifschitz, 2006], which turns the description into an equivalent definite description. The final output is an action description which can be fed into CCALC.

We asked CCALC to solve the briefcase planning problem, with the initial conditions

```
Location(Briefcase)=Home,
Holds(Briefcase,Paycheck),
Location(Dictionary)=Home,
-Holds(Briefcase,Dictionary)
```

and the goal

```
Location(Briefcase)=Office,
Location(Dictionary)=Office,
Location(Paycheck)=Home.
```

It determined that the shortest plan consists of 3 actions:

```
TakeOut(Paycheck); PutIn(Dictionary);
MoveB(Office).
```

In a different test, we instructed CCALC to display the list of all states and all transitions of the transition system represented by Figure 1, and it found 18 states and 60 transitions, as we had expected.

[5]Not to be confused with the use of the relation *Holds* in the situation calculus.

```
module MISSIONARIES;

  objects
    M1, M2, M3   : Person;
    Boat         : Vehicle;
    Bank1, Bank2 : Place;

  constants
    Board(Person),
    Disembark(Person),
    CrossTo(Place)     : action;

  variables
    m : Person;
    p : Place;

  import CARRIER;
    Load(m,Boat) is Board(m);
    Unload(m) is Disembark(m);
    Move(Boat,p) is CrossTo(p);

  axioms
    % The boat can carry at most two
    %  (i.e. not all three)
    constraint -forall m Holds(Boat,m);
```

Figure 2: Formalization of the missionaries domain

```
module AIRTRAVEL;

  objects
    George, Laura   : Person;
    AirForce1       : Vehicle;
    Austin, Lubbock : Place;

  constants
    Board(Person),
    Disembark(Person),
    Fly(Place)         : action;

  variables
    m : Person;
    p : Place;

  import CARRIER;
    Load(m,AirForce1) is Board(m);
    Unload(m) is Disembark(m);
    Move(AirForce1,p) is Fly(p);

  axioms
    % the pilot is disregarded
    %  in this formalization
    -DriverRequired(AirForce1);
```

Figure 3: Formalization of the air travel domain

All of the examples shown in this paper have been tested successfully using our system. Such tests serve to increase our confidence both in the adequacy of the formalizations and in the soundness of the implementation of MAD. We don't show any of these tests due to space constraints, though the system itself, the library modules, and all of the examples (with sample queries) are available online at `http://www.cs.utexas.edu/users/tag/mad/`.

3 The dictionary and paycheck disguised as humans

In this section we formalize two commonsense domains having to do with humans and vehicles, which are structurally very similar to the briefcase domain discussed above. One is a simplified version of the familiar missionaries and cannibals puzzle [Amarel, 1968], in which there are no cannibals—just three persons who want to cross the river, and a boat that holds two. The second, inspired by [Gelfond, 2006], involves travel by air.

In the modules `MISSIONARIES` (Figure 2) and `AIRTRAVEL` (Figure 3), sort `Person` is used in place of sort `Item` from Figure 1. There is no need for sort declarations here, because `Person` is described as a subsort of `Carrier`, and consequently a subsort of `Thing`, in the library ontology. (A person is a carrier because he can carry things in his hands or pockets. This fact will become essential in the next section.)

According to the same ontology, sort `Vehicle` is a subsort of `Carrier`. One distinctive feature of vehicles, in comparison with other carriers, is that, by default, a vehicle cannot move unless there is at least one person (driver) inside. Second, vehicles are too big to be held by people. For instance, a missionary cannot hold the boat on his back.

Unlike `BRIEFCASE`, each of the modules `MISSIONARIES` and `AIRTRAVEL` includes an axiom section. Axioms in a MAD description of a domain describe the domain-specific assumptions that are not covered by the axioms in the imported modules. In `MISSIONARIES`, the only domain-specific assumption is that the boat holds two. In `AIRTRAVEL`, we postulate that `AirForce1` is an exception to the above-mentioned default about vehicles (not because it is fully automatic, of course, but because our simplified formalization disregards the presence of a pilot). In the library module `CARRIER`, the symbol `DriverRequired` serves as the flag that can be used to disable the default about the need for a driver.

4 Takeoff and landing

Module `AIRTRAVEL_AIR` (Figure 4) is an enhancement of the air travel example that takes into account the need to take off before flying anywhere and to land after that. It imports module `AIRTRAVEL` and declares two additional actions, `TakeOff` and `Land`.

The effects of these actions are described here using the fluent `Support(x)`, declared in the library module `MOUNT`. Executing action `TakeOff` changes the value of `Support(AirForce1)` to `Air`; after executing action `Land`, its value becomes `Ground`. Both `Ground` and `Air` are objects of sort `Supporter`, which, according to the library ontology, is a supersort of sort `Thing`. `Ground`

```
module AIRTRAVEL_AIR;

  import AIRTRAVEL;

  objects
    Air : Supporter;

  constants
    TakeOff, Land : action;

  variables
    x : Thing;
    m : Person;
    p : Place;

  import MOUNT;
    Mount(AirForce1, Air) is TakeOff;

  import MOUNT;
    Mount(AirForce1, Ground) is Land;

  axioms
    % Must take off before flying
    nonexecutable Fly(p)
        if Support(AirForce1)=Ground;
    % Must land before getting in or out
    nonexecutable
        (Board(m) | Disembark(m))
          if Support(AirForce1)!=Ground;
    % Only the plane can be freely flying
    constraint
        Support(x)=Air -> x=AirForce1;
```

Figure 4: Takeoff and landing

is declared in the library; `Air` is specific for the module `AIRTRAVEL_AIR`.

In the library module `MOUNT`, the action `Mount(x,s)`, where `s` is a `Supporter`, is postulated to change the value of `Support(x)` to `s`.

5 Pednault's briefcase revisited

The enhacement of Pednault's example shown in Figure 5 takes into account the fact that the briefcase doesn't move to the office by itself; the owner carries it with him. We assume here that he walks to the office.

Module `CARRIER` is imported here twice: first, as in `BRIEFCASE`, to describe putting an item in the briefcase, and then to describe the new actions of picking up a thing and putting it down. (The action of taking an item out of the briefcase is no longer necessary in the presence of the new action `PickUp`.) The action `MoveB` from the simpler formalization is not available anymore. Instead, `Walk` is declared to be an action that changes Ed's location, and consequently the locations of all things that Ed carries.

Importing the same module twice in different ways is similar to invoking the same subroutine twice with different parameters, or to referring to a set of axioms from abstract algebra to describe first properties of addition, and then properties of multiplication in a number system.

```
sorts
  Item;

inclusions
  Item << Thing;

module BRIEFCASE_WALK;

  objects
    Ed                    : Person;
    Paycheck, Dictionary  : Item;
    Briefcase             : Carrier;
    Home, Office          : Place;

  constants
    PutIn(Item),
    PickUp(Thing),
    PutDown(Thing),
    Walk(Place)      : action;

  variables
    i : Item;
    x : Thing;
    p : Place;

  import CARRIER;
    Load(i,Briefcase) is PutIn(i);
    Unload(x) is false;
    Move(Ed,p) is Walk(p);

  import CARRIER;
    Load(x,Ed) is PickUp(x);
    Unload(x) is PutDown(x);
    Move(Ed,p) is Walk(p);

  axioms
    Smaller(Briefcase,Ed);
    nonexecutable PutDown(x)
                  if -Holds(Ed,x);
```

Figure 5: Formalization of the briefcase domain with walking

The enhanced formalization of the briefcase domain has two axioms. The first of them uses the relation `Smaller` between two things, which is introduced in module `CARRIER` for the purpose of specifying when a carrier is "too small" to hold a thing. This relation is postulated to be false by default, and we have already seen one exception to this default: humans are too small to hold vehicles. Now we postulate also that Ed Pednault's briefcase is too small to enclose its owner.

The second axiom says that Ed can put down a thing only if he is holding it.

In this formalization, Pednault's planning problem (Section 2.1) can be solved in 4 steps. He has two alternatives: after picking up the briefcase and putting the paycheck down, and before walking to the office, he can either put the dictio-

nary in the briefcase, or simply pick it up and carry it in the other hand.

6 Moving within a limited range

John McCarthy [1959] explained the fact that he needed a car to get to the airport by noting that his home and the airport do not belong to a sufficiently small, "walkable", region. They are in the same county, and counties are "drivable"—small enough to drive across. He could get to the airport by first walking to the car, which is at his home also (this is possible because his home is walkable) and then driving his car to the airport.

In the library module MOVE_IN_REGION this idea is generalized by introducing the concept of a "movable" region and postulating that the action Move(x,p) is nonexecutable unless place p lies within a movable region that contains Location(x). Region is a supersort of Place, and the inclusion relation between regions is denoted by At. By default, this relation is assumed to be false, that is to say, regions are presumed to be pairwise disjoint.

In Figure 6, MOVE_IN_REGION is used, along with CARRIER, to formalize McCarthy's example.

The logistics domain, introduced in [Veloso, 1992], is described as follows[6]:

> There are several cities, each containing several locations, some of which are airports. There are also trucks, which can drive within a single city, and airplanes, which can fly between airports. The goal is to get some packages from various locations to various new locations.

A MAD formalization of this domain is shown in Figure 7. The condition that trucks can only drive within a single city is similar to the limitations on walking and driving in McCarthy's example, and it is expressed here by importing module MOVE_IN_REGION.

This representation of logistics is abstract, in the sense that it does not declare objects corresponding to specific vehicles, places and packages. A module describing a concrete logistics domain would import module LOGISTICS, declare its objects, and provide axioms describing the At relation between places and cities.

7 Conclusion

We showed how to use the MAD language to formalize many action domains from the literature on commonsense reasoning, employing a common "standard library" of general-purpose basic actions. In particular, we showed how the library module CARRIER contains knowledge that provided the basis for representing objects as diverse as briefcases, cars, trucks, boats, airplanes, and even humans.

Using such a library allows us to abstract out the common aspects of different domains, not only making the formalizations simpler, but also helping us recognize structural similarities in domains that may seem very different at first glance.

[6]Taken from the webpage of the first International Planning Competition, ftp://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html

```
module AIRPORT;

  objects
    John                    : Person;
    Car                     : Vehicle;
    Desk, Garage, Airport   : Place;
    Home, County            : Region;

  constants
    Walkable(Region),
    Drivable(Region)   : Boolean;
    Walk(Place),
    Drive(Place),
    Board,
    Disembark          : action;

  variables
    p : Place;
    r : Region;

  import CARRIER;
    Load(John,Car) is Board;
    Unload(John) is Disembark;
    Move(Car,p) is Drive(p);

  import MOVE_IN_REGION;
    Move(Car,p) is Drive(p);
    Movable(r) is Drivable(r);

  import MOVE_IN_REGION;
    Move(John,p) is Walk(p);
    Movable(r) is Walkable(r);

  axioms
    constraint Location(Car)!=Desk;
    At(Desk, Home);
    At(Garage, Home);
    At(Home, County);
    At(Airport, County);
    Walkable(Home);
    Drivable(County);
```

Figure 6: Going to the airport

We saw that three domains, briefcase (Section 2.2), missionaries, and air travel (Section 3), had almost identical structure.

The implementation of the MAD language that we created allowed us to test our representations by calculating the sets of states of the transition systems and by making sure that the responses to reasoning queries were in accordance with our expectations.

In the future we plan to enhance the MAD language by adding the capability to reason with numbers. This will help us to extend the range of domains we can nicely represent. For example, in the Missionaries and Cannibals domain, it is preferable to reason about the number of missionaries and cannibals on each bank, rather than about individuals [Amarel, 1968].

```
sorts
  City; Airport;
  Truck; Airplane;
  Package;

inclusions
  City << Region;
  Airport << Place;
  Truck << Vehicle;
  Airplane << Vehicle;
  Package << Thing;

module LOGISTICS;

  constants
    Drivable(City)     : Boolean;
    Go(Vehicle, Place) : action;

  variables
    p : Place;
    c : City;
    v : Vehicle;
    t : Truck;
    a : Airplane;

  import CARRIER;
    Move(v,p) is Go(v,p);

  import MOVE_IN_REGION;
    Move(t,p) is Go(t,p);
    Movable(c) is Drivable(c);

  axioms
    -DriverRequired(v);
    constraint
          Location(a)=p -> Airport(p);
    Drivable(c);
```

Figure 7: Formalization of the logistics domain

There are three parts to our project of developing a library of general-purpose action descriptions: designing the MAD language, using the language to formalize new library modules along with example domains, and developing the implementation. These parts all need to be done in parallel. As we attempt to formalize more domains in simpler, better ways, we see the need for new language features and new library modules. And of course, the simultaneous development of the implementation is necessary to test the soundness and usefulness of the new language design and formalizations.

Acknowledgements

We are grateful to Michael Gelfond, Yuliya Lierler, Wanwan Ren and Yana Todorova for useful discussions related to the topic of this paper. This research was partially supported by the National Science Foundation under Grant IIS-0712113.

References

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence*, volume 3, pages 131–171. Edinburgh University Press, Edinburgh, 1968.

[Balduccini, 2007] Marcello Balduccini. CR-MODELS: An inference engine for CR-prolog. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 18–30, 2007.

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

[Doherty and Kvarnström, 2008] Patrick Doherty and Jonas Kvarnström. Temporal action logics. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*. Elsevier, 2008.

[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.

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

[Gelfond, 2006] Michael Gelfond. Going places — notes on a modular development of knowledge about travel. In *Working Notes of the AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, 2006.

[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.

[Gustafsson and Kvarnström, 2004] Joakim Gustafsson and Jonas Kvarnström. Elaboration tolerance through object-orientation. *Artificial Intelligence*, 153(1–2):239–285, 2004.

[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.

[McCarthy, 1959] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959.

[Pednault, 1988] Edwin Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

[Veloso, 1992] Manuela Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Carnegie Mellon University, 1992. PhD thesis.