A Modular Action Description Language

Vladimir Lifschitz and **Wanwan Ren**
The University of Texas at Austin
{vl,rww6}@cs.utexas.edu

Abstract

"Toy worlds" involving actions, such as the blocks world and the Missionaries and Cannibals puzzle, are often used by researchers in the areas of commonsense reasoning and planning to illustrate and test their ideas. We would like to create a database of general-purpose knowledge about actions that encodes common features of many action domains of this kind, in the same way as abstract algebra and topology represent common features of specific number systems. This paper is a report on the first stage of this project—the design of an action description language in which this database will be written. The new language is an extension of the action language $\mathcal{C}+$. Its main distinctive feature is the possibility of referring to other action descriptions in the definition of a new action domain.

Introduction

Since the early days of AI, researchers have been using "toy worlds" to illustrate and test their ideas related to action and change. For instance, a large part of the classical paper (Amarel 1968) is about the Missionaries and Cannibals puzzle; the answer literal method is explained in (Green 1969) using two other domains: Monkey and Bananas and the Towers of Hanoi. An attempt to use circumscription to solve the frame problem is described in (McCarthy 1986) in terms of the blocks world, and the need to modify that approach is demonstrated in (Hanks & McDermott 1987) using the Yale Shooting scenario. Many "benchmark problems" of this kind are collected in (Mueller 2006, Table 1.1). Toy worlds are used also in planning competitions.

These action domains have many common features, and we would like to create a database of general facts related to actions that encodes these features, just like abstract algebra and topology represent common properties of specific mathematical structures. When a mathematician describes axioms for the system of real numbers, he can say that it is a group relative to addition; by doing this, he replaces a part of the axiom set with a reference to a general concept. In the same way, we would like to avoid the need to list explicitly some of the properties of crossing the river in the Missionaries and Cannibals puzzle, climbing on the box in the Monkey and Bananas domain, stacking blocks in the blocks world, and so forth. Instead, we will be able to refer to the abstract concept of moving an object, described in the database of general-purpose knowledge about actions.

This paper is a preliminary report on the first stage of this project—on the design of the language in which the database will be written. Our proposal is a step towards "getting a language for expressing general common sense knowledge for inclusion in a general database" that can be used by any program that needs the knowledge, which is "the key problem of generality in AI" (McCarthy 1987).

The new *M*odular *A*ction *D*escription language, or MAD, is based on the action language $\mathcal{C}+$ (Giunchiglia *et al.* 2004) but differs from it in that a MAD action description generally consists of several modules $M_1, \ldots, M_n$ that may contain references to other modules in this list.[1] A module M_i can use, or "import," any of the modules $M_1, \ldots, M_{i-1}$, 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 import construct is essential for our purposes, because descriptions of specific action domains will need to import parts of the general database. MAD will also help us organize the database in a hierarchical way, with modules of a more general nature imported by more specialized modules.

Erdoğan & Lifschitz (2006) argue that the ability to define more specific kinds of actions in terms of more general kinds is important because this is what human often do when they describe actions informally. For instance, the dictionary explains *walk* as "move by foot," and *climb* as "go up or down." These explanations of the words *walk* and *climb* do not list the effects of these

[1]The use of the word "modular" in this paper is not related to its use in (Herzig & Varzinczak 2004) and (Kakas, Michael, & Miller 2005).

actions; rather, they present these actions as special cases of some other actions that are supposed to be already familiar to us. The most fundamental actions still need to be described directly in terms of the changes that they cause. The word *move*, for instance, means "cause to change position," according to the dictionary. But in many cases the best way to describe an action is to relate it to something more general. The language MAD allows us to do this on the basis of a precisely defined semantics.

We expect that MAD will prove itself more elaboration tolerant (McCarthy 2003) than nonmodular action languages: it will be relatively easy to modify a set of facts expressed in MAD to take into account new phenomena or changed circumstances.

Reusable modules in the context of declarative programming and knowledge representation have been discussed by many authors; see, for instance, (Bugliesi, Lamma, & Mello 1994), (Amir 1999), (Barker, Porter, & Clark 2001), (Gustafsson & Kvarnström 2004), (Ianni *et al.* 2004). Some of this work is applicable to describing actions. For example, the Component Library (`http://www.cs.utexas.edu/users/mfkb/RKF/clib.html`) describes the action *Transfer* by its add-list and delete-list, in the spirit of STRIPS (Fikes & Nilsson 1971); then *Lose* is described as the special case of *Transfer* characterized by an additional axiom. MAD differs from this earlier work in three ways.

First, its semantics, like the semantics of other action languages (Gelfond & Lifschitz 1998), is defined in terms of particularly simple, well-understood mathematical objects—transition diagrams of the kind familiar from automata theory. Models of a MAD action description are directed graphs whose vertices are states, and whose edges are labeled by events (such as Figure 2 in the next section).[2]

Second, the syntax and semantics of MAD are based on ideas of nonmonotonic causal logic (Geffner 1990; McCain & Turner 1997; Giunchiglia *et al.* 2004), which is much more expressive than STRIPS and its successor ADL (Pednault 1994). In MAD we can describe actions with indirect effects and indirect preconditions, nondeterministic actions, and non-serializable concurrently executed actions.

Third, because of the close relationship between action languages and answer sets (Lifschitz & Turner 1999), it will be possible to solve many computational problems for action domains described in MAD, such as planning, using answer set programming systems (Lifschitz 2002).

Example

Figure 1 shows a MAD action description consisting of two modules. Module MOVE is an abstract axiomatization of "move-like" actions; it can be included, in principle, in a general-purpose database. (The version of MOVE that we actually plan to include in the database is more sophisticated in a number of ways.) Module MONKEY uses MOVE to describe a simplified form of

[2]The only other attempt to define a modular action language that we are aware of is outlined in (Clark, Porter, & Batory 1996), where STRIPS operators are formed from "components," similar to modules in the sense of this paper.

module MOVE;

  sorts
    Thing; *Place*;

  constants
    Location(*Thing*): **fluent**(*Place*);
    Move(*Thing*,*Place*): **action**;

  variables
    x: *Thing*; p: *Place*;

  axioms
    inertial $Location(x)$;
    exogenous $Move(x,p)$;
    $Move(x,p)$ **causes** $Location(x) = p$;
    nonexecutable $Move(x,p)$ **if** $Location(x) = p$;

endmodule;

module MONKEY;

  sorts
    Thing; *Place*; *Level*;

  objects
    Monkey, *Box*: *Thing*;
    P_1, P_2: *Place*;
    BoxTop, *Floor*: *Level*;

  constants
    OnBox: **fluent**;
    Walk(*Place*), *ClimbOn*, *ClimbOff*: **action**;

  variables
    x: *Thing*; p: *Place*; l: *Level*;

  import MOVE;
    $Move(x,p)$ **is** $Walk(p) \wedge x = Monkey$;

  import MOVE;
    Place **is** *Level*;
    $Location(x) = l$ **is**
        $((x = Monkey \wedge OnBox) \wedge l = BoxTop) \vee$
        $(\neg(x = Monkey \wedge OnBox) \wedge l = Floor)$;
    $Move(x,l)$ **is**
        $(x = Monkey \wedge l = BoxTop \wedge ClimbOn) \vee$
        $(x = Monkey \wedge l = Floor \wedge ClimbOff)$;

  axioms
    $Location(Monkey) = p$
        if $OnBox \wedge Location(Box) = p$;
    nonexecutable *ClimbOn*
        if $Location(Monkey) \neq Location(Box)$;
    nonexecutable $ClimbOff \wedge Walk(p)$;

endmodule

Figure 1: A MAD action description

the Monkey and Bananas domain.

This section is an informal discussion of the main parts of this action description.

Module MOVE

According to the sort declaration at the beginning of the module, there are objects of two kinds: things and places. According to the constant declarations, the location of a thing is a place-valued fluent, and moving a thing to a place is an action.

The four axioms at the end of the module are similar to causal laws of action language $\mathcal{C}+$ (Giunchiglia *et al.* 2004, Section 4.2). The location of an object is inertial: it does not change without a cause. The move actions are exogenous: they can be executed or not executed at will. Moving an object affects its location. An object cannot be moved to its current location.

Module MONKEY

This module describes a simplified version of the Monkey and Bananas domain that includes the monkey and the box, but not bananas; furthermore, the box cannot be moved. The monkey can walk to another place and can climb on and off the box. All three actions are treated here as special cases of *Move*.

There are objects of three kinds: things, places and levels. The monkey and the box are things; Place 1 and Place 2 are places; the top of the box and the floor are levels. Being on the box is a truth-valued fluent. Walking to a place, climbing on the box and climbing off the box are actions.

The variable declarations are followed by two import statements. Both of them import the module MOVE, but in different ways, because actions in this domain can be viewed as instances of *Move* in two ways.

The first import statement

$$\begin{array}{l}\textbf{import}\ \mathrm{MOVE};\\ \quad \mathit{Move}(x,p)\ \textbf{is}\ \mathit{Walk}(p) \wedge x = \mathit{Monkey};\end{array} \qquad (1)$$

tells us that the axioms from the module MOVE hold if we understand $\mathit{Move}(x,p)$ as follows:

the action $\mathit{Walk}(p)$ is executed, and x is *Monkey*.

According to this special interpretation of *Move*, the only thing that can move is the monkey, and the name for this special case of *Move* is *Walk*.

The second import statement describes moving "along the vertical axis." The constant *Move* is reinterpreted again; also, the sort *Place* and the constant *Location* are reinterpreted. In this import statement, places are understood as levels, so that we can talk about moving from the floor to the top of the box and the other way around. Furthermore, $\mathit{Location}(x)$ is understood

- as *BoxTop* if x is Monkey and *OnBox* holds, and
- as *Floor* otherwise.

Finally, $\mathit{Move}(x,l)$ can be executed in two ways:

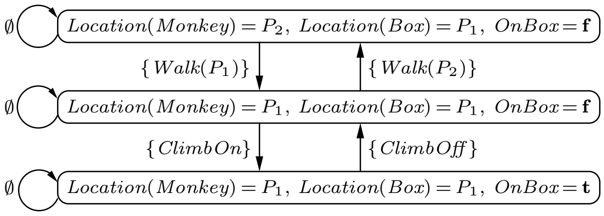

Figure 2: A half of the standard model of the action description shown in Figure 1. The other half can be obtained by swapping P_1 and P_2. There are no edges connecting the two parts of the graph with each other, because no action in this domain changes the location of the box.

- by executing *ClimbOn* if x is *Monkey* and l is *BoxTop*, and
- by executing *ClimbOff* if x is *Monkey* and l is *Floor*.

The three axioms at the end of module MONKEY express the additional properties of the domain that have no counterparts in our general theory of move-like actions. First, whenever the monkey is on the box, he is at the same place where the box is.[3] Second, the monkey can't climb on the box if the box is at a different place. Finally, he can't climb off and walk to another place simultaneously. (Other pairs of actions cannot be executed concurrently either, as follows from the axioms already included.)

Standard model

According to the semantics of MAD described below, the "standard model" of the action description in Figure 1 is a directed graph with 6 vertices, corresponding to the possible combinations of values of the fluents

$$\mathit{Location}(\mathit{Monkey}),\ \mathit{Location}(\mathit{Box}),\ \mathit{OnBox}.$$

The graph has 6 vertices rather than 8 because the first two fluents cannot have different values if *OnBox* is true. Three of these vertices, along with the edges connecting them, are shown in Figure 2. Every edge is labeled by a set of action constants, which is either a singleton or the empty set, because in this example no two actions can be executed concurrently. The edges labeled $\emptyset$ are self-loops, because in this example a state does not change unless an action is executed.

Syntax of MAD

The structure of an action description

As can be seen from the example in Figure 1, a MAD action description is a list of modules. A module includes a name and several optional parts, in a fixed order: sort declarations, object declarations, constant declarations,

[3]The use of the $\mathcal{C}+$ construct **if** in this axiom instead of material implication conveys the idea that there is a causal relationship between the location of the box and the location of the monkey (Giunchiglia *et al.* 2004, Section 3.2).

variable declarations, and axioms. Additionally, import statements may appear anywhere between these parts.

A sort name is usually an identifier, although it may also consist of several identifiers separated by dots; similarly for object names, constant names, and variable names. Names consisting of several identifiers are used as auxiliary syntactic expressions in the definition of the semantics of MAD below.

Axioms are expressions similar to causal laws in the sense of $\mathcal{C}+$ (Giunchiglia *et al.* 2004, Section 4.2 and Appendix B). Thus the list of axioms of a MAD module is essentially a $\mathcal{C}+$ action description (to be more precise, it becomes a $\mathcal{C}+$ action description after grounding).

An import statement consists of the reserved word **import** and the name of the imported module; it may also include sort renaming clauses and constant renaming clauses:

<import statement>
  ::= **import** <module name>';'
    {<sort renaming clause>';'}
    {<constant renaming clause>';'}
<sort renaming clause>
  ::= <sort name> **is** <sort name>
<constant renaming clause>
  ::= <constant name> ['('{<variable name>','}
    <variable name>')'] ['='<variable name>]
    is <formula>

For instance, the second import statement in Figure 1 contains one sort renaming clause and two constant renaming clauses.

Context-dependent conditions

In an action description that consists of modules $M_1, \ldots, M_n$, the names of the modules M_i should be different from each other. For every import statement *IS* occurring in M_i, its module name should be the name of one of the modules M_j with $j < i$; we will say that *IS refers to* this module M_j.

The condition that a name should not be declared more than once, and that a name should not be used unless it has been declared earlier, applies to modules in a MAD action description with two caveats.

First, a name can be declared in a module not only explicitly, but also implicitly—in an import statement. For instance, the first import statement in module MONKEY (Figure 1) implicitly declares *Thing* and *Place* to be sort names, and *Location* to be a constant name. (But it does not declare *Move*; see below.)

To give the general definition of "implicitly declared," note first that any import statement has the form

$$\begin{array}{l} \textbf{import } \mathit{NAME}; \\ \quad s_1 \textbf{ is } s'_1; \\ \quad \cdots \\ \quad s_k \textbf{ is } s'_k; \\ \quad c_1 \ \cdots \ \textbf{is } F_1; \\ \quad \cdots \\ \quad c_l \ \cdots \ \textbf{is } F_l; \end{array} \tag{2}$$

where *NAME* is a module name, $s_1, \ldots, s_k, s'_1, \ldots, s'_k$ are sort names, $c_1, \ldots, c_l$ are constant names. (The dots after each c_j represent the two optional parts in the rule for <constant renaming clause> above.) We define the relation "implicitly declares" recursively, as follows. An occurrence of an import statement (2) in M_i *implicitly declares* a name z to be a sort name (or object name, or constant name) if

(i) z is declared to be a sort name (respectively, object name or constant name), explicitly or implicitly, in the module that (2) refers to, and

(ii) z is different from $s_1, \ldots, s_k, c_1, \ldots, c_l$.

Note that, according to this definition, a variable name cannot be declared implicitly; each variable name in MAD is "local" to the module in which it is declared. Condition (ii) expresses that $s_1, \ldots, s_k, c_1, \ldots, c_l$ are "renamed" in the specialization of the module that (2) refers to. For instance, *Move* is not declared, even implicitly, in module MONKEY (Figure 1).

For any import statement (2) occurring in M_i, the names $s_1, \ldots, s_k, c_1, \ldots, c_l$ should be pairwise distinct, and they should be declared, explicitly or implicitly, in the module that (2) refers to. Every other name used in M_i should be declared in M_i, explicitly or implicitly, before it is used.

Second, multiple declarations of the same name in a module are allowed as long as at most one of these declarations is explicit and all of them declare the name in the same way. For instance, the sort name *Thing* is declared three times in the second module of Figure 1—explicitly at the beginning of the module, and implicitly twice by the two import statements.

Additional conditions on constant renaming clauses

A constant renaming clause, according to the definition above, has the form

$$c(x_1, \ldots, x_p) \textbf{ is } F \tag{3}$$

or

$$c(x_1, \ldots, x_p) = y \textbf{ is } F \tag{4}$$

(without the parentheses if $p = 0$). In either case, we require that

- the variables to the left of **is** be pairwise distinct, and
- every free variable of F occur to the left of **is**.

In a constant renaming clause of the form (3), constant c should be Boolean.

In a constant renaming clause of the form (4), variable y should be of the same sort as constant c. Furthermore, F should "define y in terms of $x_1, \ldots, x_p$" in the sense that the formula

$$\forall x_1 \cdots x_p \exists y' \forall y (F \equiv y = y')$$

is universally valid. For instance, the constant renaming clause beginning with $\mathit{Location}(x) = l$ in Figure 1 satisfies this condition because the formula

$$\forall x \exists l' \forall l ((((x = \mathit{Monkey} \land \mathit{OnBox}) \land l = \mathit{BoxTop}) \lor (\neg(x = \mathit{Monkey} \land \mathit{OnBox}) \land l = \mathit{Floor})) \equiv l = l')$$

is universally valid.

Semantics of MAD

The semantics of MAD is defined here by translating MAD into $\mathcal{C}+$. The semantics of $\mathcal{C}+$ action descriptions is described in (Giunchiglia *et al.* 2004, Sections 4.2, 4.4).

As a preliminary step, we show how to convert an arbitrary MAD action description into a single module. For instance, the single-module action description corresponding to Figure 1 is shown in Figure 3. Note the names beginning with I1 and I2 in this module. Intuitively, I1.*Move* and I2.*Move* are the "copies" of the constant *Move* from module MOVE that correspond to the two import statements in Figure 1.

Then we show how to translate any single-module action description into $\mathcal{C}+$ by grounding.

Generating a single-module description

We begin by defining three auxiliary functions. The function α turns a module that does not contain import statements into its "specialized form" in accordance with a given import statement. The function β "merges" two modules. The function γ eliminates the first import statement from a given action description.

Let M be a module without import statements, IS an import statement (2) such that $NAME$ is the name of M, and m a positive integer. By $\alpha(M, IS, m)$ we denote the module obtained from M by

- replacing every occurrence of each of the sort names s_i with s'_i $(i = 1, \ldots, k)$;
- prepending "Im." to every occurrence of every variable name and to every occurrence of each of the constant names c_j $(j = 1, \ldots, l)$;
- inserting the equivalences

$$\mathrm{I}m.c_j \ \ldots \ \equiv F_j, \quad (j = 1, \ldots, l), \tag{5}$$

  corresponding to the constant renaming clauses from (2), at the beginning of the axiom part.[4]

An example is shown in Figure 4.

Let M and M' be modules such that M' does not contain import statements. By $\beta(M, M')$ we denote the module obtained by appending

- the sort names declared in M' but not in M,
- the object, constant and variable declarations from M', with repetitions removed, and
- the axioms of M'

[4]These equivalences are similar to rules for lifting statements to other contexts in (McCarthy 1993), and to $\mathcal{C}+$ bridge rules studied in (Erdoğan & Lifschitz 2006).

module MONKEY;

  sorts
    Thing; *Place*; *Level*;

  objects
    Monkey, *Box*: *Thing*;
    P_1, P_2: *Place*;
    BoxTop, *Floor*: *Level*;

  constants
    OnBox: **fluent**;
    Walk(*Place*), *ClimbOn*, *ClimbOff*: **action**;
    Location(*Thing*): **fluent**(*Place*);
    I1.*Move*(*Thing*,*Place*): **action**;
    I2.*Location*(*Thing*): **fluent**(*Level*);
    I2.*Move*(*Thing*,*Level*): **action**;

  variables
    x: *Thing*; p: *Place*; l: *Level*;
    I1.x: *Thing*; I1.p: *Place*;
    I2.x: *Thing*; I2.p: *Level*;

  axioms
    $\mathit{Location}(\mathit{Monkey}) = p$
      if $\mathit{OnBox} \land \mathit{Location}(\mathit{Box}) = p$;
    nonexecutable *ClimbOn*
      if $\mathit{Location}(\mathit{Monkey}) \neq \mathit{Location}(\mathit{Box})$;
    nonexecutable $\mathit{ClimbOff} \land \mathit{Walk}(p)$;

    $\mathrm{I1}.\mathit{Move}(x, p) \ \equiv \ \mathit{Walk}(p) \land x = \mathit{Monkey}$;
    inertial $\mathit{Location}(\mathrm{I1}.x)$;
    exogenous $\mathrm{I1}.\mathit{Move}(\mathrm{I1}.x, \mathrm{I1}.p)$;
    $\mathrm{I1}.\mathit{Move}(\mathrm{I1}.x, \mathrm{I1}.p)$ **causes** $\mathit{Location}(\mathrm{I1}.x) = \mathrm{I1}.p$;
    nonexecutable $\mathrm{I1}.\mathit{Move}(\mathrm{I1}.x, \mathrm{I1}.p)$
      if $\mathit{Location}(\mathrm{I1}.x) = \mathrm{I1}.p$;

    $\mathrm{I2}.\mathit{Location}(x) = l$
      $\equiv ((x = \mathit{Monkey} \land \mathit{OnBox}) \land l = \mathit{BoxTop}) \lor$
      $(\neg(x = \mathit{Monkey} \land \mathit{OnBox}) \land l = \mathit{Floor})$;
    $\mathrm{I2}.\mathit{Move}(x, l)$
      $\equiv (x = \mathit{Monkey} \land l = \mathit{BoxTop} \land \mathit{ClimbOn}) \lor$
      $(x = \mathit{Monkey} \land l = \mathit{Floor} \land \mathit{ClimbOff})$;
    inertial $\mathrm{I2}.\mathit{Location}(\mathrm{I2}.x)$;
    exogenous $\mathrm{I2}.\mathit{Move}(\mathrm{I2}.x, \mathrm{I2}.p)$;
    $\mathrm{I2}.\mathit{Move}(\mathrm{I2}.x, \mathrm{I2}.p)$
      causes $\mathrm{I2}.\mathit{Location}(\mathrm{I2}.x) = \mathrm{I2}.p$;
    nonexecutable $\mathrm{I2}.\mathit{Move}(\mathrm{I2}.x, \mathrm{I2}.p)$
      if $\mathrm{I2}.\mathit{Location}(\mathrm{I2}.x) = \mathrm{I2}.p$;

endmodule

Figure 3: A single module corresponding to Figure 1

to the corresponding parts of M.

Let $M_1; \ldots; M_n$ be an action description containing at least one import statement. By $\gamma(M_1; \ldots; M_n)$ we denote the action description obtained by replacing M_i with

$$\beta(M, \alpha(M_j, IS, m))$$

where

module MOVE;

sorts
Thing; *Place*;

constants
Location(*Thing*): **fluent**(*Place*);
I1.*Move*(*Thing*,*Place*): **action**;

variables
I1.x: *Thing*; I1.p: *Place*;

axioms
I1.$\mathit{Move}(x,p) \equiv \mathit{Walk}(p) \wedge x = \mathit{Monkey}$;

inertial $\mathit{Location}(\mathrm{I1}.x)$;
exogenous $\mathrm{I1}.\mathit{Move}(\mathrm{I1}.x, \mathrm{I1}.p)$;
$\mathrm{I1}.\mathit{Move}(\mathrm{I1}.x, \mathrm{I1}.p)$ **causes** $\mathit{Location}(\mathrm{I1}.x) = \mathrm{I1}.p$;
nonexecutable $\mathrm{I1}.\mathit{Move}(\mathrm{I1}.x, \mathrm{I1}.p)$
if $\mathit{Location}(\mathrm{I1}.x) = \mathrm{I1}.p$;

endmodule

Figure 4: The result of applying α to module MOVE from Figure 1 and import statement (1), with $m = 1$

- M_i is the first module in $M_1; \ldots; M_n$ that contains an import statement,
- *IS* is the first import statement in M_i,
- M is the module obtained from M_i by dropping *IS*,
- M_j is the module that *IS* refers to,
- m is the smallest positive integer such that the string "Im." does not occur in $M_1; \ldots; M_n$.

It is clear that applying γ to an action description decrements the number of import statements by 1. If $M_1; \ldots; M_n$ contains p import statements then $\gamma^p(M_1; \ldots; M_n)$ is an action description $M'_1; \ldots; M'_n$ that does not contain import statements. We denote its last term M'_n by $\delta(M_1; \ldots; M_n)$. This is the single-module action description that we consider to have the same meaning as $M_1; \ldots; M_n$.

For instance, the result of applying translation δ to Figure 1 is Figure 3.

Grounding

Let M be a single-module action description. A *universe function* for M is a function U that assigns a finite nonempty set of symbols to each sort name s of M so that $U(s)$ contains all object names of sort s but no other names declared in M. We call $U(s)$ the *universe* of sort s. For instance, a universe function for Figure 3 can be defined by

$$\begin{array}{l} U(\mathit{Thing}) = \{\mathit{Monkey}, \mathit{Box}\}, \\ U(\mathit{Place}) = \{P_1, P_2\}, \\ U(\mathit{Level}) = \{\mathit{BoxTop}, \mathit{Floor}\}. \end{array} \tag{6}$$

The result of *grounding* M relative to a universe function U is the $\mathcal{C}+$ action description[5] M_U that is formed according to the following rules.

The signature σ of M_U is determined by the constant declaration part of M, as follows. If the constant declaration part contains a constant declaration

$$\ldots, c(s_1, \ldots, s_k), \ldots : \textbf{fluent}(s)$$

then σ includes the symbols $c(z_1, \ldots, z_k)$ for all $z_1 \in U(s_1), \ldots, z_k \in U(s_k)$, designated as simple fluent constants with the domain $U(s)$. If **fluent** in the constant declaration is not followed by (s) then the corresponding constants in σ are Boolean. If the declaration contains the symbol **action** instead of **fluent** then the corresponding symbols in σ are action constants.

For instance, if M is the action description in Figure 3, and U is defined by (6), then σ consists of 5 simple fluent constants and 12 Boolean action constants:

OnBox, $\mathit{Location}(\mathit{Monkey})$, $\mathit{Location}(\mathit{Box})$,
$\mathrm{I2}.\mathit{Location}(\mathit{Monkey})$, $\mathrm{I2}.\mathit{Location}(\mathit{Box})$,
$\mathit{Walk}(P_1)$, $\mathit{Walk}(P_2)$, $\mathit{ClimbOn}$, $\mathit{ClimbOff}$,
$\mathrm{I1}.\mathit{Move}(\mathit{Monkey}, P_1)$, $\mathrm{I1}.\mathit{Move}(\mathit{Box}, P_1)$,
$\mathrm{I1}.\mathit{Move}(\mathit{Monkey}, P_2)$, $\mathrm{I1}.\mathit{Move}(\mathit{Box}, P_2)$,
$\mathrm{I2}.\mathit{Move}(\mathit{Monkey}, \mathit{BoxTop})$, $\mathrm{I2}.\mathit{Move}(\mathit{Box}, \mathit{BoxTop})$,
$\mathrm{I2}.\mathit{Move}(\mathit{Monkey}, \mathit{Floor})$, $\mathrm{I2}.\mathit{Move}(\mathit{Box}, \mathit{Floor})$.

The causal laws of M_U are obtained from the axioms of M by substituting for each variable v arbitrary elements of $U(s)$, where s is the sort assigned to v in the variable declaration part of M.

According to the semantics of MAD, the model of an action description D corresponding to a universe function U for $\delta(D)$ is the transition system represented by the $\mathcal{C}+$ action description $\delta(D)_U$, as defined in the semantics of $\mathcal{C}+$ (Giunchiglia *et al.* 2004, Section 4.4).

If for every sort name s declared in $\delta(D)$ the set of object names of sort s in $\delta(D)$ is nonempty then the function that maps every sort name s to the set of object names of sort s is a universe function. The model of D corresponding to this universe function is the *standard model* of D. For instance, the standard model of the action description in Figure 1, which is depicted in Figure 2, corresponds to the universe function characterized by formulas (6). To be precise, the labels of vertices and edges of the standard model in this example contain, besides the symbols shown in Figure 2, several auxiliary constants beginning with I1 and I2.

Conclusion

The language MAD, outlined in this paper, serves for describing action domains. A MAD action description consists of modules that may contain references to other modules. Semantically, such a description represents a family of transition systems.

We plan to use MAD to compile a general-purpose database of knowledge about actions, in which we will "factor out" common elements of specific action domains studied in the literature on commonsense reasoning and planning. In the process, we will identify constructs that can be added to the basic version of MAD described above to make it a more versatile tool.

[5]For the definition of a $\mathcal{C}+$ action description see (Giunchiglia *et al.* 2004, Section 4.2).

We plan also to extend the Causal Calculator[6] to domains described in MAD. The new version of CCalc will be used to test and refine the general-purpose database of knowledge about actions. Unlike the current version, it will need to handle some nondefinite causal laws (laws with non-atomic heads), because the equivalences (5), introduced by translation δ described above, are non-definite.

Acknowledgements

We are grateful to Chitta Baral, Peter Clark, Selim Erdoğan, Paolo Ferraris, Michael Gelfond, Joohyung Lee, John McCarthy, Bruce Porter and Hudson Turner for useful discussions, and to the anonymous referees for their comments. This work was partially supported by the National Science Foundation under Grant IIS-0412907.

References

Amarel, S. 1968. On representations of problems of reasoning about actions. In Michie, D., ed., *Machine Intelligence*, volume 3. Edinburgh: Edinburgh University Press. 131–171.

Amir, E. 1999. Object-oriented first-order logic.[7] *Electronic Transactions on Artificial Intelligence* 4:63–84.

Barker, K.; Porter, B.; and Clark, P. 2001. A library of generic concepts for composing knowledge bases. In *First International Conference on Knowledge Capture*, 14–21.

Bugliesi, M.; Lamma, E.; and Mello, P. 1994. Modularity in logic programming. *Journal of Logic Programming* 19/20:443–502.

Clark, P.; Porter, B.; and Batory, D. 1996. A compositional approach to representing planning operators.[8] Technical Report AI06-331, University of Texas at Austin.

Erdoğan, S. T., and Lifschitz, V. 2006. Actions as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*. To appear.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3–4):189–208.

Geffner, H. 1990. Causal theories for nonmonotonic reasoning. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 524–530. AAAI Press.

Gelfond, M., and Lifschitz, V. 1998. Action languages.[9] *Electronic Transactions on Artificial Intelligence* 3:195–210.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence* 153(1–2):49–104.

Green, C. 1969. Application of theorem proving to problem solving. In Walker, D., and Norton, L., eds., *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 219–240. The MITRE Corporation.

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

Hanks, S., and McDermott, D. 1987. Nonmonotonic logic and temporal projection. *Artificial Intelligence* 33(3):379–412.

Herzig, A., and Varzinczak, I. 2004. Domain descriptions should be modular. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, 348–352.

Ianni, G.; Ielpa, G.; Pietramala, A.; Santoro, M. C.; and Calimeri, F. 2004. Enhancing answer set programming with templates. In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, 233–239.

Kakas, A.; Michael, L.; and Miller, R. 2005. Modular-E: an elaboration tolerant approach to the ramification and qualification problems. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 211–226.

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

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

McCain, N., and Turner, H. 1997. Causal theories of action and change. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 460–465.

McCarthy, J. 1986. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence* 26(3):89–116. Reproduced in (McCarthy 1990).

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

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

McCarthy, J. 1993. Notes on formalizing context. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 555–560.

McCarthy, J. 2003. Elaboration tolerance.[10] In progress.

Mueller, E. 2006. *Commonsense reasoning.* Elsevier.

Pednault, E. 1994. ADL and the state-transition model of action. *Journal of Logic and Computation* 4:467–512.

[6] The Causal Calculator (CCalc) is a planning and query answering system for action domains described in $\mathcal{C}+$ (`http://www.cs.utexas.edu/users/tag/ccalc/`).

[7] `http://www.ep.liu.se/ea/cis/1999/042/` .

[8] `http://www.cs.utexas.edu/ftp/pub/AI-Lab/tech-reports/UT-AI-TR-06-331.pdf` .

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

[10] `http://www-formal.stanford.edu/jmc/elaboration.html` .