

# Towards a Modular Action Description Language

Vladimir Lifschitz and Wanwan Ren

University of Texas at Austin  
{vl, rww6}@cs.utexas.edu

## Abstract

This is a preliminary report on the design of a modular language for describing actions. In the new language, an action description consists of several modules; each module describes a set of interrelated fluents and actions. “Import statements” allow the user to provide references to other modules and thus characterize new fluents and actions by relating them to others, introduced earlier. This capability is essential for designing a repository of background knowledge involving actions, because descriptions of specific action domains will need to “import” parts of the repository.

## Introduction

This is a preliminary report on the design of a new language for describing actions. This Modular Action Description 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  $M_1, \dots, M_n$  that may contain references to other modules. 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 new expressive capabilities of MAD are essential for designing a repository of background knowledge involving actions, because descriptions of specific action domains will need to import parts of the repository. MAD will also help us organize a repository of background knowledge in a hierarchical way, with modules of a more general nature imported by more specialized modules.

Erdoğan and Lifschitz (2005) argue that 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 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 will allow us to do this on the basis of a precisely defined semantics.

Besides actions, some authors introduce “strategies,” or “programs” (McCarthy and Hayes 1969; Levesque *et al.* 1998), which are formed from actions using operations similar to the programming constructs found in procedural programming languages (sequential and parallel composition, loop, etc.) Actions are characterized in terms of their behavior; strategies, in terms of their structure. In this note we limit attention to actions and do not discuss strategies.

Reusable modules in the context of declarative programming and knowledge representation have been discussed by many authors; see, for instance, (Bugliesi *et al.* 1994), (Amir 1999), (Barker *et al.* 2001), (Gustafsson and Kvarnström 2004), (Ianni *et al.* 2004). This paper is different from the earlier work in that its goal is to define a language for describing transition systems—directed graphs whose vertices are states, and whose edges are labeled by events.

Assuming that the reader has some familiarity with action language  $\mathcal{C}+$ , we discuss here a series of examples illustrating the main elements of the syntax and semantics of MAD and the applicability of the new language to the problem of compiling a general-purpose database of knowledge about fluents and actions. The last part of the paper outlines the syntax and semantics of a fragment of MAD.

## Single-module action descriptions

### MAD as a superset of $\mathcal{C}+$

MAD is a superset of  $\mathcal{C}+$  in the sense that any action description in  $\mathcal{C}+$  can be easily rewritten as a MAD

---

```

module SUITCASE;

  constants
    Up, Open: fluent;
    Toggle: action;

  axioms
    inertial Up, Open;
    exogenous Toggle;
    Toggle causes Up if  $\neg$ Up;
    Toggle causes  $\neg$ Up if Up;
    Open if Up;

endmodule

```

---

Figure 1: Module SUITCASE

module. Take as an example the simplified version of the suitcase domain from (Lin 1995) in which the suitcase has one latch, rather than two; the latch can be toggled, and whenever it is in the up position, the suitcase opens. This domain can be described in  $\mathcal{C}+$  using the Boolean action constant<sup>1</sup> *Toggle* and the simple Boolean fluent constants<sup>2</sup> *Up* and *Open*, as follows:<sup>3</sup>

```

inertial Up, Open
exogenous Toggle
Toggle causes Up if  $\neg$ Up
Toggle causes  $\neg$ Up if Up
caused Open if Up

```

According to the semantics of  $\mathcal{C}+$  (Giunchiglia *et al.* 2004, Section 4.4), this action description represents a transition system—a directed graph whose vertices are states, and whose edges are labeled by events. Each state assigns a truth value to the fluent constants *Up* and *Open*; there are only 3 states, because *Open* cannot be false when *Up* is true. Each event assigns a truth value to the action constant *Toggle*, so that 2 events are possible: toggle the latch or do nothing.

A translation of this action description into MAD is shown in Figure 1. It differs from the  $\mathcal{C}+$  description in three ways. First, as any MAD module, it has a name; giving a name to a module allows us to refer to this module in import statements. Second, besides causal laws (grouped under the heading **axioms**), it contains declarations of action and fluent constants; in this respect, MAD is similar to the input language of the system C<sub>CALC</sub>, which implements  $\mathcal{C}+$ .<sup>4</sup> Third, we

<sup>1</sup>Recall that the syntax of  $\mathcal{C}+$  includes also action constants with non-Boolean values (Giunchiglia *et al.* 2004, Sections 4.2, 5.6).

<sup>2</sup>The word “simple” distinguishes these symbols from statically determined fluent constants (Giunchiglia *et al.* 2004, Sections 4.2, 4.4).

<sup>3</sup>See (Giunchiglia *et al.* 2004, Appendix B) for an explanation of most abbreviations used in this example. We also combine here two expressions of the form **inertial** *c* into one.

<sup>4</sup><http://www.cs.utexas.edu/users/tag/ccalc/> .

---

```

module SUITCASE-V;

  sorts
    Latch;

  constants
    Up(Latch), Open: fluent;
    Toggle(Latch): action;

  variables
    l: Latch;

  axioms
    inertial Up(l), Open;
    exogenous Toggle(l);
    Toggle(l) causes Up(l) if  $\neg$ Up(l);
    Toggle(l) causes  $\neg$ Up(l) if Up(l);
    Open if  $\forall l$  Up(l);

endmodule

```

---

Figure 2: Module SUITCASE-V

simplified the syntax of the last causal law by dropping the word **caused**; this difference is not essential.

## Variables in MAD

Module SUITCASE-V (Figure 2) uses a variable of sort *Latch* to describe suitcases with an arbitrary number of latches. Note the use of a quantifier in the last causal law of SUITCASE-V to express that the suitcase is open whenever all latches are up.

This module represents a family of transition systems (“models”), not a single system. According to the semantics of MAD, SUITCASE-V tells us how to turn any finite non-empty set *L* of symbols into a transition system in which *L* plays the role of the set of all objects of the sort *Latch*.<sup>5</sup> This transition system has  $2^{n+1} - 1$  states, where *n* is the cardinality of *L*; the states correspond to all assignments of truth values to the fluent constants *Up*(*l*) (*l* ∈ *L*) and *Open* except for the one that is eliminated by the last causal law of SUITCASE-V. Each of its edges is labeled by one of  $2^n$  events—by an assignment of truth values to the action constants *Toggle*(*l*) (*l* ∈ *L*). If *L* is a singleton then the corresponding model of SUITCASE-V is isomorphic to the model of SUITCASE.

## Importing a module

### Import statements

A simple example of the use of import statements in MAD is shown in Figures 3 and 4. Module LATCH is a simplified version of SUITCASE-V, which describes the effect of *Toggle*(*Latch*) on *Up*(*Latch*) but does not mention the fluent constant *Open*. Module SUITCASE-V'

<sup>5</sup>In the semantics of MAD, the sets representing sorts are always non-empty, just as the universe of a model in the semantics of first-order logic is non-empty.

---

```

module LATCH;

  sorts
    Latch;

  constants
    Up(Latch): fluent;
    Toggle(Latch): action;

  variables
    l: Latch;

  axioms
    inertial Up(l);
    exogenous Toggle(l);
    Toggle(l) causes Up(l) if  $\neg Up(l)$ ;
    Toggle(l) causes  $\neg Up(l)$  if Up(l);

endmodule

```

---

Figure 3: Module LATCH

---

```

module SUITCASE-V';

  import LATCH;

  constants
    Open: fluent;

  variables
    l: Latch;

  axioms
    inertial Open;
    Open if  $\forall l Up(l)$ ;

endmodule

```

---

Figure 4: Module SUITCASE-V'

imports LATCH and adds the declaration of *Open* along with the causal laws from Figure 2 in which *Open* is mentioned. The action description

$$\text{LATCH SUITCASE-V'} \quad (1)$$

has the same meaning as SUITCASE-V.

Note that SUITCASE-V' includes a declaration of a variable for latches, even though such a variable is already declared in LATCH. This declaration is necessary because in MAD each variable is considered local to the module in which it is declared; the variable *l* from LATCH is not “visible” in SUITCASE-V'.

As another example of the use of import statements in MAD, consider the action description

$$\text{LATCH SUITCASE-V' SUITCASE-V}'_2. \quad (2)$$

The module SUITCASE-V'<sub>2</sub>, shown in Figure 5, imports SUITCASE-V' and then declares *L*<sub>1</sub> and *L*<sub>2</sub> to be objects of sort *Latch*. Action description (2) describes suitcases that have at least the two latches *L*<sub>1</sub>, *L*<sub>2</sub>. The models of (2) can be characterized as the models of

---

```

module SUITCASE-V}'_2;

  import SUITCASE-V';

  objects
    L1, L2: Latch;

endmodule

```

---

Figure 5: Module SUITCASE-V'<sub>2</sub>

SUITCASE-V corresponding to the finite sets *L* such that *L*<sub>1</sub>, *L*<sub>2</sub> ∈ *L*.

In the “standard” model of (2), *L* = {*L*<sub>1</sub>, *L*<sub>2</sub>}. The standard model of a MAD action description is, intuitively, the model in which every sort has the smallest possible extent. The only model of the action description SUITCASE (Figure 1) is trivially standard, because SUITCASE does not contain sort declarations. Action description (1) does not have a standard model, because its sort *Latch* is not “populated”: the smallest possible extent of *Latch* is empty. (Recall that, in any MAD model, every sort is represented by a non-empty finite set.) No MAD action description can have more than one standard model.

The functionality of the future implementation of MAD will allow the user to get answers to queries about the standard model of a given action description. In the standard model, quantifiers (such as  $\forall l$  in Figure 2) can be equivalently replaced by finite conjunctions and disjunctions. In view of this fact, many queries about the standard model of a MAD action description can be answered by invoking a propositional satisfiability solver or an answer set solver, such as SMOBELS.<sup>6</sup>

### Turning an action description into a single module

The semantics of MAD specifies, for any action description *M*<sub>1</sub> ··· *M*<sub>*n*</sub>, what its models are, and which model (if any) is standard. The basic form of the semantics defines the models and the standard model for the action descriptions that consist of a single module, such as SUITCASE and SUITCASE-V. This definition is extended to arbitrary action descriptions using a procedure for rewriting an action description consisting of several modules as one module. This procedure eliminates all import statements from given action description; each of them is replaced by a modified copy of the imported module.

As an example, consider the application of this procedure to action description (1). The first—and only—import statement in (1) is

$$\text{import LATCH} \quad (3)$$

in module SUITCASE-V'. The result of the elimination process, shown in Figure 6, is the module obtained from SUITCASE-V' by removing this import statement and

<sup>6</sup><http://www.tcs.hut.fi/Software/smodels/> .

---

```

module SUITCASE-V';

  sorts
    Latch;

  constants
    Open: fluent;
    Up(Latch): fluent;
    Toggle(Latch): action;

  variables
    l: Latch;
    I1.l: Latch;

  axioms
    inertial Open;
    Open if  $\forall l Up(l)$ ;
    inertial Up(I1.l);
    exogenous Toggle(I1.l);
    Toggle(I1.l) causes Up(I1.l) if  $\neg Up(I1.l)$ ;
    Toggle(I1.l) causes  $\neg Up(I1.l)$  if Up(I1.l);

endmodule

```

---

Figure 6: A single module corresponding to action description (1)

including instead the contents of module LATCH, as follows:

- the sort declaration part of LATCH is included in the new module without change;
- the constant declaration part of LATCH is merged with the constant declaration part of SUITCASE-V';
- the variable declaration part of LATCH is merged with the variable declaration part of SUITCASE-V' after renaming the variable  $l$  in LATCH—prepending to it  $I1$ , which stands for “Import 1”;
- the list of causal laws from LATCH is merged with the list of causal laws from SUITCASE-V' after renaming the variables in LATCH as above.

Generally, to turn an action description  $M_1 \dots M_n$  into a single module, we eliminate all import statements from  $M_2$ , then from  $M_3$ , and so on, using distinct symbols, one per import statement, to rename variables. After eliminating all import statements from  $M_n$ , the modules  $M_1, \dots, M_{n-1}$  are dropped. For instance, to apply this procedure to (2), we first replace the second term of (2) by the version of SUITCASE-V' shown in Figure 6, then eliminate

**import** SUITCASE-V'

from the last term of (2), and then drop the first two terms, which have now become irrelevant. The result is shown in Figure 7. (Placing object declarations after sort declarations and before constant declarations is required by the syntax of MAD.)

---

```

module SUITCASE-V'_2;

  sorts
    Latch;

  objects
    L1, L2: Latch;

  constants
    Open: fluent;
    Up(Latch): fluent;
    Toggle(Latch): action;

  variables
    I2.l: Latch;
    I2.I1.l: Latch;

  axioms
    inertial Open;
    Open if  $\forall I2.l Up(I2.l)$ ;
    inertial Up(I2.I1.l);
    exogenous Toggle(I2.I1.l);
    Toggle(I2.I1.l) causes Up(I2.I1.l)
      if  $\neg Up(I2.I1.l)$ ;
    Toggle(I2.I1.l) causes  $\neg Up(I2.I1.l)$ 
      if Up(I2.I1.l);

endmodule

```

---

Figure 7: A single module corresponding to action description (2)

### Importing background knowledge

We are interested in import statements because in descriptions of action domains they may be used as references to a repository of background knowledge.

As an example, consider the module in Figure 8, which provides an abstract description of the action *Get* and its effect on the fluent *Has*. A module like this may serve as part of a general-purpose repository of knowledge. A description of a specific action domain can import GET and provide additional information about the domain by “populating” the sorts *Agent* and *Thing*, for instance:

```

import GET;

objects
  Surgeon: Agent;
  Scalpel1, Scalpel2, Scalpel3: Thing;

```

### Specialization

We would like to use import statements to define more specific kinds of actions in terms of more general kinds. For instance, we may wish to use a module describing general properties of moving things to characterize a special kind of moving, say walking or climbing. There is no reason to include the whole general theory of moving in the modules that describe walking and climbing; what such a specialized module needs to import is only a “special case” of the general theory.

---

```

module GET;

  sorts
    Agent; Thing;

  constants
    Has(Agent, Thing): fluent;
    Get(Agent, Thing): action;

  variables
    a, a': Agent;
    x: Thing;

  axioms
    inertial Has(a, x);
    exogenous Get(a, x);
     $\neg$ Has(a', x) if Has(a, x)  $\wedge$  a  $\neq$  a';
    Get(a, x) causes Has(a, x);

endmodule

```

---

Figure 8: Module GET

---

```

module SCALPEL;

  sorts
    Agent; Thing;

  objects
    Surgeon: Agent;
    Scalpel1, Scalpel2, Scalpel3: Thing;

  constants
    GetScalpel: action;

  variables
    a: Agent;
    x: Thing;

  import GET;
    Get(a, x) is
      GetScalpel  $\wedge$  a = Surgeon  $\wedge$  x = Scalpel3;

endmodule

```

---

Figure 9: Module SCALPEL

The syntax of MAD allows us to say which special case of a module is imported. For instance, in Figure 9 we import the special case of the module GET that deals with the effect of the action  $Get(a, x)$  when  $a = Surgeon$  and  $x = Scalpel3$ . This action is given the name  $GetScalpel$ ; <sup>7</sup> the other actions of the form  $Get(a, x)$  cannot be even referred to in the module SCALPEL. Syntactically, this is achieved by including the “**is** clause”

$Get(a, x)$  **is**  $GetScalpel \wedge a = Surgeon \wedge x = Scalpel3$   
at the end of the import statement in Figure 9.

<sup>7</sup>In the context of an operation, when the surgeon says, “Scalpel”, this may be equivalent to the sentence, “Please give me the number 3 scalpel” (McCarthy 1993).

---

```

module SCALPEL;

  sorts
    Agent; Thing;

  objects
    Surgeon: Agent;
    Scalpel1, Scalpel2, Scalpel3: Thing;

  constants
    GetScalpel: action;
    Has(Agent, Thing): fluent;
    I1.Get(Agent, Thing): action;

  variables
    a: Agent;
    x: Thing;
    I1.a, I1.a': Agent;
    I1.x: Thing;

  axioms
    I1.Get(a, x)  $\equiv$ 
      GetScalpel  $\wedge$  a = Surgeon  $\wedge$  x = Scalpel3;
    inertial Has(I1.a, I1.x);
    exogenous I1.Get(I1.a, I1.x);
     $\neg$ Has(I1.a', I1.x) if Has(I1.a, I1.x)  $\wedge$  I1.a  $\neq$  I1.a';
    I1.Get(I1.a, I1.x) causes Has(I1.a, I1.x);

endmodule

```

---

Figure 10: A single module corresponding to action description (4)

The result of eliminating the import statement from the action description

$$\text{GET SCALPEL} \quad (4)$$

is shown in Figure 10. After merging the sort declaration sections of the modules GET and SCALPEL, each of the sorts  $Agent$ ,  $Thing$  is declared twice, and the repetitions are removed. The presence of an **is** clause in the import statement affects the translation process in two ways. First, the constant  $Get$  is renamed: it becomes  $I1.Get$ . Renamed constants from an imported module, like variables declared in an imported module, are “invisible”—they cannot be referred to. Second, an additional causal law is included—an equivalence expressing that the left-hand side of the **is** clause is synonymous to its right-hand side. The use of causal laws of this kind to express synonymy is discussed in (Erdoğan and Lifschitz 2005, Section 2).

### Importing a module several times

Next we consider a general-purpose module MOVE (Figure 11), describing the effect of the action  $Move(x, p)$  (“move object  $x$  to place  $p$ ”), and use that module to describe a domain that involves moves of three different kinds. This is a simplified version of the familiar Monkey and Bananas domain that includes the monkey and the box, but not bananas; furthermore,

---

```

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;

endmodule

```

---

Figure 11: Module MOVE

the box cannot be moved. The monkey can walk to another place and can climb on and off the box. All these actions are viewed as special cases of *Move*. In all cases, the object that moves is the monkey; on the other hand, walking changes the monkey’s “horizontal location,” and climbing changes his “vertical location.” Figure 12 uses module MOVE twice to represent all these actions as special cases of the “library action” *Move*.

To distinguish between the two meanings of the word “location” in the monkey domain, we declare a new sort, *VPlace*. In the second import statement, not only the constants *Move* and *Location* are renamed, but also a sort: here “place” means “vertical place.”

The single module corresponding to the action description

$$\text{MOVE MONKEY} \quad (5)$$

is shown in Figure 13.

## Syntax of MAD

In this section we describe a partial syntax of MAD that covers all examples discussed above.

### A context-free grammar

An action description in the MAD language is a list of modules:

$$\langle \text{action description} \rangle ::= \{ \langle \text{module} \rangle \} \langle \text{module} \rangle$$

A module consists of a name and a body, enclosed between the reserved words **module** and **endmodule**:

$$\langle \text{module} \rangle ::= \mathbf{module} \langle \text{module name} \rangle \text{ ; } \\ \langle \text{module body} \rangle \mathbf{endmodule}$$

The body of a module contains several parts, in a fixed order: sort declarations, object declarations, constant declarations, variable declarations and axioms. Additionally, import statements may appear anywhere between these parts:

---

```

module MONKEY;

  sorts
    Thing; Place; VPlace;

  objects
    Monkey, Box: Thing;
    P1, P2: Place;
    BoxTop, Floor: VPlace;

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

  variables
    x: Thing;
    p: Place;
    vp: VPlace;

  import MOVE;
    Move(x,p) is Walk(p) ∧ x = Monkey;

  import MOVE;
    Place is VPlace;
    Location(x) = vp is
      (x = Monkey ∧ vp = BoxTop ∧ OnBox) ∨
      (x = Monkey ∧ vp = Floor ∧ ¬OnBox) ∨
      (x = Box ∧ vp = Floor);
    Move(x, vp) is
      (x = Monkey ∧ vp = BoxTop ∧ ClimbOn) ∨
      (x = Monkey ∧ vp = Floor ∧ ClimbOff);

  axioms
    Location(Monkey) = p
      if OnBox ∧ Location(Box) = p;
    nonexecutable Walk(p) if OnBox;
    nonexecutable ClimbOn
      if Location(Monkey) ≠ Location(Box);
    nonexecutable ClimbOn ∧ Walk(p);

endmodule

```

---

Figure 12: Module MONKEY

$$\langle \text{module body} \rangle ::= \{ \langle \text{import statement} \rangle \\ \langle \text{sort declaration part} \rangle \\ \langle \text{import statement} \rangle \\ \langle \text{object declaration part} \rangle \\ \langle \text{import statement} \rangle \\ \langle \text{constant declaration part} \rangle \\ \langle \text{import statement} \rangle \\ \langle \text{variable declaration part} \rangle \\ \langle \text{import statement} \rangle \\ \langle \text{axiom part} \rangle \\ \langle \text{import statement} \rangle \}$$

The sort declaration part consists of the reserved word **sorts** and a list of sort names:

$$\langle \text{sort declaration part} \rangle ::= \mathbf{sorts} \{ \langle \text{sort name} \rangle \text{ ; } \}$$

$$\langle \text{sort name} \rangle ::= \langle \text{name} \rangle$$

The object declaration part consists of the reserved

---

```

module MONKEY;

sorts
  Thing; Place; VPlace;

objects
  Monkey, Box: Thing;
  P1, P2: Place;
  BoxTop, Floor: VPlace;

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

variables
  x: Thing;
  p: Place;
  vp: VPlace;
  I1.x: Thing;
  I1.p: Place;
  I2.x: Thing;
  I2.p: VPlace;

axioms
  Location(Monkey) = p
    if OnBox  $\wedge$  Location(Box) = p;
  nonexecutable Walk(p) if OnBox;
  nonexecutable ClimbOn
    if Location(Monkey)  $\neq$  Location(Box);
  nonexecutable ClimbOn  $\wedge$  Walk(p);
  I1.Move(x, p)  $\equiv$  Walk(p)  $\wedge$  x = Monkey;
  inertial Location(I1.x);
  exogenous I1.Move(I1.x, I1.p);
  I1.Move(I1.x, I1.p) causes Location(I1.x) = I1.p;
  I2.Location(x) = vp
     $\equiv$  (x = Monkey  $\wedge$  vp = BoxTop  $\wedge$  OnBox)  $\vee$ 
      (x = Monkey  $\wedge$  vp = Floor  $\wedge$   $\neg$ OnBox)  $\vee$ 
      (x = Box  $\wedge$  vp = Floor);
  I2.Move(x, vp)
     $\equiv$  (x = Monkey  $\wedge$  vp = BoxTop  $\wedge$  ClimbOn)  $\vee$ 
      (x = Monkey  $\wedge$  vp = Floor  $\wedge$  ClimbOff);
  inertial I2.Location(I2.x);
  exogenous I2.Move(I2.x, I2.p);
  I2.Move(I2.x, I2.p)
    causes I2.Location(I2.x) = I2.p;

endmodule

```

---

Figure 13: A single module corresponding to action description (5)

word **objects** and a list of object specifications. An object specification is a list of object names followed by a sort name:

```

<object declaration part>
  ::= objects {<object spec>','}
<object spec>
  ::= {<object name>','} <object name> ':'
    <sort name>
<object name> ::= <name>

```

The constant declaration part consists of the reserved word **constants** and a list of constant specifications. A constant specification shows, for each constant that is declared, the number of arguments that it takes and their sorts. It shows also whether the constants represent actions or fluents, and what the sort of the value is:

```

<constant declaration part>
  ::= constants {<constant spec>','}
<constant spec>
  ::= {<constant expr>','} <constant expr> ':'
    <constant type> ['(<sort name>')']
<constant expr>
  ::= <constant name>
    ['(' {<sort name>','} <sort name> ')']
<constant type> ::= action | fluent
<constant name> ::= <name>

```

The form of the variable declaration part is similar to the form of the object declaration part:

```

<variable declaration part>
  ::= variables {<variable spec>','}
<variable spec>
  ::= {<variable name>','} <variable name> ':'
    <sort name>
<variable name> ::= <name>

```

The axiom part consists of the reserved word **axioms** and a list of causal laws:

```

<axiom part> ::= axioms { <causal law> ',' }
<causal law> ::= <static law> |
  <action dynamic law> |
  <fluent dynamic law>
<static law>
  ::= <fluent formula> [if <fluent formula>]
<action dynamic law>
  ::= <action formula> [if <formula>] |
    exogenous <action constant>
<fluent dynamic law>
  ::= <fluent formula> [if <fluent formula>]
    [after <formula>] |
    <action formula> causes <fluent formula>
    [if <formula>] |
    nonexecutable <action formula>
    [if <formula>] |
    inertial <fluent constant>
<action constant>
  ::= <constant name>
    ['(' {<argument>','} <argument> ')']

```

$\langle \text{fluent constant} \rangle$   
 $::= \langle \text{constant name} \rangle$   
 $[\langle \text{'\{<argument>'>\}' <argument>'>} \rangle]$   
 $\langle \text{argument} \rangle ::= \langle \text{variable name} \rangle \mid$   
 $\langle \text{object name} \rangle$

We skip here the syntax of formulas, and turn to import statements. 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:

$\langle \text{import statement} \rangle$   
 $::= \text{import } \langle \text{module name} \rangle \langle \text{'>'} \rangle$   
 $\{ \langle \text{sort renaming clause} \rangle \langle \text{'>'} \rangle \}$   
 $\{ \langle \text{constant renaming clause} \rangle \langle \text{'>'} \rangle \}$   
 $\langle \text{sort renaming clause} \rangle$   
 $::= \langle \text{sort name} \rangle \text{ is } \langle \text{sort name} \rangle$   
 $\langle \text{constant renaming clause} \rangle$   
 $::= \langle \text{constant name} \rangle [\langle \text{'\{<variable name>'>\}' <variable name>'>} \rangle]$   
 $[\langle \text{'='>'} \langle \text{variable name} \rangle \rangle]$   
 $\text{is } \langle \text{formula} \rangle$

A name is an identifier or a list of identifiers separated by dots:

$\langle \text{name} \rangle ::= \{ \langle \text{identifier} \rangle \langle \text{'.'} \rangle \} \langle \text{identifier} \rangle$

## Context-dependent conditions

In an action description

$$M_1 \cdots M_n$$

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 MAD action descriptions with three caveats.

First, this condition applies within each module  $M_i$  separately. For instance, using an identifier as a sort name in  $M_1$  and as a constant name in  $M_2$  is not considered a syntax error (at least if  $M_1$  is not imported by  $M_2$ ; see below).

Second, a name can be declared not only explicitly—in a declaration, but also implicitly—in an import statement. For instance, import statement (3) in the second module of action description (1) implicitly declares *Latch* to be a sort name and *Up*, *Toggle* to be constant names.

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

**import**  $NAME$ ;  
 $s_1$  **is**  $s'_1$ ;  
 $\dots$   
 $s_k$  **is**  $s'_k$ ;  
 $c_1 \cdots$  **is**  $F_1$ ;  
 $\dots$   
 $c_l \cdots$  **is**  $F_l$ ;  
(6)

where  $NAME$  is a module name,  $s_1, \dots, s_k$  are sort names, and  $c_1, \dots, c_l$  are constant names. (The dots after each  $c_j$  represent the two optional parts in the rule for  $\langle \text{constant renaming clause} \rangle$  above.) We define the relation “implicitly declared” recursively, as follows. An occurrence of an import statement (6) 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 (6) refers to, and
- (ii)  $z$  is different from  $s_1, \dots, s_k, c_1, \dots, c_l$ .

Note that, according to this definition, a variable name cannot be declared implicitly; as we have seen, each variable name in MAD is “local” to the module in which it is declared. Clause (ii) expresses that  $s_1, \dots, s_k, c_1, \dots, c_l$  are “renamed” in the specialization of  $M_j$  that is described by (6). For instance, *Get* is not declared, even implicitly, in the second module of action description (4). (The name  $II.Get$  is not declared in that module either, even though it is declared and used in the corresponding single module shown in Figure 10.)

For any import statement (6) occurring in  $M_i$ , the names  $s_1, \dots, s_k, c_1, \dots, c_l$  should be declared, explicitly or implicitly, in the module that (6) refers to. Otherwise, each name used in any of the modules  $M_i$  should be declared in  $M_i$ , explicitly or implicitly, before it is used.

Third, 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 names *Agent* and *Thing* are declared twice in the second module of action description (4)—explicitly at the beginning of the module, and implicitly at the end.

## Semantics of MAD

The first part of the semantics of MAD shows how to turn an arbitrary action description into a single-module action description that is considered to have the same meaning. The second part describes the process of grounding, which turns an arbitrary single-module action description into a family of action descriptions in the language  $\mathcal{C}+$ .

## Generating a single-module action description

We begin by defining three auxiliary functions. The function  $\alpha$  turns a module  $M$  that does not contain import statements into its “specialized form” in accordance with an import statement  $IS$ . The function  $\beta$  “merges” a module  $M$  with a module  $M'$  that does not contain import statements. (Here we understand the

---

```

module GET;
  sorts
    Agent; Thing;
  constants
    Has(Agent, Thing): fluent;
    I1.Get(Agent, Thing): action;
  variables
    I1.a, I1.a': Agent;
    I1.x: Thing;
  axioms
    I1.Get(a, x)  $\equiv$ 
      GetScalpel  $\wedge$  a = Surgeon  $\wedge$  x = Scalpel3;
    inertial Has(I1.a, I1.x);
    exogenous I1.Get(I1.a, I1.x);
     $\neg$ Has(I1.a', I1.x) if Has(I1.a, I1.x)  $\wedge$  I1.a  $\neq$  I1.a';
    I1.Get(I1.a, I1.x) causes Has(I1.a, I1.x);
endmodule

```

---

Figure 14: The result of applying  $\alpha$  to the module shown in Figure 8 and to the import statement from Figure 9, with  $m = 1$

word “module” in the sense of the context-free grammar above.) The function  $\gamma$  eliminates the first import statement from a given action description.

Without loss of generality, we can assume that every module under consideration includes each of the reserved words

**sorts objects constants variables axioms**

—they can be inserted at appropriate places, if necessary.

Let  $M$  be a module without import statements,  $IS$  an import statement (6) 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

- (i) prepending ‘ $Im.$ ’ to every occurrence of every variable name;
- (ii) replacing every occurrence of each of the sort names  $s_i$  with  $s'_i$  ( $i = 1, \dots, k$ );
- (iii) inserting the equivalences

$$c_j \dots \equiv F_j$$

( $j = 1, \dots, l$ ), corresponding to the constant renaming clauses from (6), at the beginning of the axiom part;

- (iv) prepending ‘ $Im.$ ’ to every occurrence of each of the constant names  $c_j$  ( $j = 1, \dots, l$ ).

See Figure 14 for an example.

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

- (i) appending the sort names declared in  $M'$  at the end of the sort declaration part and removing repetitions;
- (ii) appending the object specifications from  $M'$  at the end of the object declaration part and removing repetitions;
- (iii) appending the constant specifications from  $M'$  at the end of the constant declaration part and removing repetitions;
- (iv) appending the variable specifications from  $M'$  at the end of the variable declaration part and removing repetitions;
- (v) appending the causal laws from  $M'$  at the end of the axiom part.

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

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

where

- $M_i$  is the first module in  $M_1 \dots M_n$  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  $M_1 \dots M_n$ .

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

Consider, for example, the application of  $\delta$  to action description (4). In this case,  $n = 2$ ,  $M_1$  is the module shown in Figure 8, and  $M_2$  is the module shown in Figure 9. This action description contains one import statement  $IS$ , so that  $p = 1$  and  $\delta(M_1 M_2)$  is the second module in  $\gamma(M_1 M_2)$ . According to the definition of  $\gamma$ , that module is

$$\beta(M, \alpha(M_1, IS, 1)),$$

where the first argument  $M$  is obtained from Figure 9 by dropping the import statement, and the second argument is shown in Figure 14. We have seen the result of this computation in Figure 10.

## Grounding

Recall that a (*multi-valued propositional*) *signature* is a set  $\sigma$  of symbols called *constants*, along with a nonempty finite set  $Dom(c)$  of symbols (the *domain* of  $c$ ), disjoint from  $\sigma$ , assigned to each constant  $c$  (Giunchiglia *et al.* 2004, Section 2.1). A *formula* of  $\sigma$  is

a propositional combination of *atoms* of the form  $c = v$  where  $c \in \sigma$  and  $v \in \text{Dom}(c)$ . To specify a  $\mathcal{C}+$  action description, we first choose a signature, partitioned into *simple fluent constants*, *statically determined fluent constants*, and *action constants*; then we specify a set of causal laws formed from formulas of this signature (Giunchiglia *et al.* 2004, Section 4.2).

Take a single-module action description  $M$ . 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 the action description shown in Figure 10 is defined by a finite “universe of agents”  $U(\text{Agent})$  and a finite “universe of things”  $U(\text{Thing})$  such that

$$\begin{aligned} \text{Surgeon} &\in U(\text{Agent}), \\ \text{Scalpel1}, \text{Scalpel2}, \text{Scalpel3} &\in U(\text{Thing}). \end{aligned}$$

Relative to a universe function  $U$ ,  $M$  has the same meaning as a  $\mathcal{C}+$  action description  $M_U$ , which is formed according to several simple rules, as follows.

The signature  $\sigma$  of  $M_U$  is determined by the constant declaration part of  $M$ , as follows. If the constant declaration part contains a constant specification

$$\dots, c, \dots : \mathbf{fluent}(s)$$

then  $\sigma$  includes the simple fluent constant  $c$  with the domain  $U(s)$ . If the constant declaration part contains a constant specification

$$\dots, c(s_1, \dots, s_k), \dots : \mathbf{fluent}(s)$$

then  $\sigma$  includes the symbols  $c(z_1, \dots, z_k)$  for all  $z_1 \in U(s_1), \dots, z_k \in U(s_k)$ , designated as simple fluent constants with the domain  $U(s)$ . If the constant type **fluent** in the constant specification is not followed by  $(s)$  then the corresponding constants in  $\sigma$  are considered Boolean (that is, their domain is  $\{\mathbf{f}, \mathbf{t}\}$ ). If the constant type is **action** rather than **fluent** then the corresponding symbols in  $\sigma$  are action constants. (The fragment of MAD discussed in this paper does not allow us to declare statically determined fluent constants.)

For instance, if  $M$  is the action description shown in Figure 10, and the universe function  $U$  is defined by the formulas

$$\begin{aligned} U(\text{Agent}) &= \{\text{Surgeon}\}, \\ U(\text{Thing}) &= \{\text{Scalpel1}, \text{Scalpel2}, \text{Scalpel3}\}, \end{aligned} \quad (7)$$

then  $\sigma$  consists of the Boolean simple fluent constants

$$\begin{aligned} \text{Has}(\text{Surgeon}, \text{Scalpel1}), \text{Has}(\text{Surgeon}, \text{Scalpel2}), \\ \text{Has}(\text{Surgeon}, \text{Scalpel3}) \end{aligned}$$

and the Boolean action constants

$$\begin{aligned} \text{GetScalpel}, \text{I1.Get}(\text{Surgeon}, \text{Scalpel1}), \\ \text{I1.Get}(\text{Surgeon}, \text{Scalpel2}), \text{I1.Get}(\text{Surgeon}, \text{Scalpel3}). \end{aligned}$$

The causal laws of  $M_U$  are obtained from the causal laws in the axiom part of  $M$  by grounding; the symbols substituted for each variable  $v$  in the process of

grounding are arbitrary elements of  $U(s)$ , where  $s$  is the sort assigned to  $v$  in the variable declaration part of  $M$ . Quantifiers (as, for instance, in the second causal law of Figure 7) are replaced by conjunctions and disjunctions.

According to the semantics of MAD, the model of an action description  $D$  corresponding to a universe function  $U$  of  $\delta(D)$  is the transition system represented by the  $\mathcal{C}+$  action description  $\delta(D)_U$ .

Assume that for every sort name  $s$  declared in  $\delta(D)$  the set of object names of sort  $s$  in  $\delta(D)$  is nonempty. In this case, 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 action description (4) corresponds to the universe function characterized by formulas (7).

## Conclusion

A MAD action description consists of modules that include causal laws in the sense of  $\mathcal{C}+$  and may also contain references to other modules. It defines a family of transition systems; at most one of them is “standard.”

A module can be used to encode knowledge about actions of a general nature, such as *Get* and *Move*. A collection of modules of this kind may serve as a repository of background knowledge that complements the domain-specific information provided in knowledge representation and reasoning problems involving actions.

Our plans for the future include adding a few more useful features to the fragment of MAD presented above, implementing MAD on the basis of a satisfiability solver or an answer set solver, and using the new language to compile a general-purpose database of knowledge about actions.

## Acknowledgements

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

## References

- Eyal Amir. Object-oriented first-order logic. *Electronic Transactions on Artificial Intelligence*, 3C:63–84, 1999.
- Ken Barker, Bruce Porter, and Peter Clark. A library of generic concepts for composing knowledge bases. In *First International Conference on Knowledge Capture*, 2001.
- Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.
- Selim T. Erdoğan and Vladimir Lifschitz. Actions as special cases (preliminary report). In *Working Notes of IJCAI-05 Workshop on Nonmonotonic Reasoning, Action, and Change*, pages 34–38, 2005.

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

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

Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, Maria Carmela Santoro, and Francesco Calimeri. Enhancing answer set programming with templates. In *Proceedings of the 10th International Workshop on Nonmonotonic Reasoning (NMR-04)*, pages 233–239, 2004.

Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus.<sup>8</sup> *Electronic Transactions on AI*, 2:159–178, 1998.

Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1985–1991, 1995.

John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

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

---

<sup>8</sup><http://www.ep.liu.se/ea/cis/1998/018/> .