Yet Another Modular Action Language

Michael Gelfond and Daniela Inclezan

Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
Michael.Gelfond@ttu.edu, daniela.inclezan@ttu.edu

Abstract. The paper presents an action language, $\mathcal{ALM}$, for the representation of knowledge about dynamic systems. It extends action language $\mathcal{AL}$ by allowing definitions of new objects (actions and fluents) in terms of other, previously defined, objects. This, together with the modular structure of the language, leads to more elegant and concise representations and facilitates the creation of libraries of knowledge modules.

1 Introduction

This paper presents an extension, $\mathcal{ALM}$, of action language $\mathcal{AL}$ [1], [2] by simple but powerful means for describing modules. $\mathcal{AL}$ is an action language used for the specification of dynamic systems which can be modeled by transition diagrams whose nodes correspond to possible physical states of the domain and whose arcs are labeled by actions. It has a developed theory, methodology of use, and a number of applications [3]. However, it lacks the structure needed for expressing the hierarchies of abstractions often necessary for the design of larger knowledge bases and the creation of KR-libraries. The goal of this paper is to remedy this problem. System descriptions of our new language, $\mathcal{ALM}$, are divided into two parts. The first part contains *declarations* of the sorts, fluents, and actions of the language. Intuitively, it defines an uninterpreted theory of the system description. The second part, called *structure*, gives an interpretation of this theory by defining particular instances of sorts, fluents, and actions relevant to a given domain. Declarations are divided into *modules* organized as tree-like hierarchies. This allows for actions and fluents to be defined in terms of other actions and fluents. For instance, the action *carry* (defined in a dictionary as "to move while supporting") can be declared as a special case of *move*. There are two other action languages with modular structure. Language MAD [4],[5] is an expansion of action language $\mathcal{C}$ [6]. Even though $\mathcal{C}$ and $\mathcal{AL}$ have a lot in common, they differ significantly in the underlying assumptions incorporated in their semantics. For example, the semantics of $\mathcal{AL}$ incorporates the *inertia axiom* [7] which says that "*Things normally stay the same.*" The statement is a typical example of a default, which is to a large degree responsible for the very close and natural connections between $\mathcal{AL}$ and ASP [8]. $\mathcal{C}$ is based on a different assumption – the so called *causality principle* – which says that "*Everything true in the world must be caused.*" Its underlying logical basis is causal logic [9].

There is also a close relationship between ASP and $\mathcal{C}$ but, in our judgment, the distance between ASP and $\mathcal{ALM}$ is much smaller than that between ASP and $\mathcal{C}$. Another modular language is TAL-C [10], which allows definitions of classes of objects that are somewhat similar to those in $\mathcal{ALM}$. TAL-C, however, seems to have more ambitious goals: the language is used to describe and reason about various dynamic scenarios, whereas in $\mathcal{ALM}$ the description of a scenario and that of reasoning tasks are not viewed as part of the language.

The differences in the underlying languages and in the way structure is incorporated into $\mathcal{ALM}$, MAD and TAL-C lead to very different knowledge representation styles. We believe that this is a good thing. Much more research and experience of use is needed to discover if one of these languages has some advantages over the others, or if different languages simply correspond to and enhance different habits of thought.

This paper consists of two parts. First we define the syntax and semantics of an auxiliary extension of $\mathcal{AL}$ by so called defined fluents. The resulting language, $\mathcal{AL}_d$, will then be expanded to $\mathcal{ALM}$.

2 Expanding $\mathcal{AL}$ by Defined Fluents

2.1 Syntax of $\mathcal{AL}_d$

A *system description* of $\mathcal{AL}_d$ consists of a sorted *signature* and a collection of *axioms*. The signature contains the names for primitive *sorts*, a *sorted universe* consisting of non-empty sets of object constants assigned to each such name, and names for *actions* and *fluents*. The fluents are partitioned into *statics*, *inertial fluents*, and *defined fluents*. The truth values of statics cannot be changed by actions. Inertial fluents can be changed by actions and are subject to the law of inertia. Defined fluents are non-static fluents which are defined in terms of other fluents. They can be changed by actions but only indirectly. An atom is a string of the form $p(\bar{x})$ where p is a fluent and $\bar{x}$ is a tuple of primitive objects. A *literal* is an atom or its negation. Depending on the type of fluent forming a literal we will use the terms *static*, *inertial*, and *defined literal*. We assume that for every sort s and constant c of this sort the signature contains a static, $s(c)$. Direct causal effects of actions are described in $\mathcal{AL}_d$ by *dynamic causal laws* – statements of the form:

$$a \textbf{ causes } l \textbf{ if } p \tag{1}$$

where l is an inertial literal, a is an action name, and p is a collection of arbitrary literals. (1) says that if action a were executed in a state satisfying p then l would be true in a state resulting from this execution. Dependencies between fluents are described by *state constraints* — statements of the form:

$$l \textbf{ if } p \tag{2}$$

where l is a literal and p is a set of literals. (2) says that every state satisfying p must satisfy l. *Executability conditions* of $\mathcal{AL}_d$ are statements of the form:

$$\textbf{impossible } a_1, \dots, a_k \textbf{ if } p \tag{3}$$

The statement says that actions $a_1, \dots, a_k$ cannot be executed together in any state which satisfies p. We refer to l as the head of the corresponding rule and to p as its body. The collection of state constraints whose head is a defined fluent f is referred to as the *definition of* f. As in logic programming definitions, f is true in a state σ if the body of at least one of its defining constraints is true in σ. Otherwise, f is false. Finally, an expression of the form

$$f \equiv g \textbf{ if } p \tag{4}$$

where f and g are inertial or static fluents and p is a set of literals, will be understood as a shorthand for four state constraints:

f **if** p, g $\quad \neg f$ **if** $p, \neg g$ $\quad g$ **if** p, f $\quad \neg g$ **if** $p, \neg f$

An $\mathcal{AL}_d$ axiom with variables is understood as a shorthand for the set of all its ground instantiations.

2.2 Semantics of $\mathcal{AL}_d$

To define the semantics of $\mathcal{AL}_d$, we define the transition diagram $\mathcal{T}(\mathcal{D})$ for every system description $\mathcal{D}$ of $\mathcal{AL}_d$. Some preliminary definitions: a set σ of literals is called *complete* if for any fluent f either f or $\neg f$ is in σ; σ is called *consistent* if there is no f such that $f \in \sigma$ and $\neg f \in \sigma$. Our definition of the transition relation $\langle \sigma_0, a, \sigma_1 \rangle$ of $\mathcal{T}(\mathcal{D})$ *will be based on the notion of an answer set of a logic program.* We will construct a program $\Pi(\mathcal{D})$ consisting of logic programming encodings of statements from $\mathcal{D}$. The answer sets of the union of $\Pi(\mathcal{D})$ with the encodings of a state σ_0 and an action a will determine the states into which the system can move after the execution of a in σ_0.

The signature of $\Pi(\mathcal{D})$ will contain: (a) names from the signature of $\mathcal{D}$; (b) two new sorts: *steps* with two constants, 0 and 1, and *fluent_type* with constants *inertial*, *static*, and *defined*; and (c) the relations: $holds(fluent, step)$ ($holds(f, i)$ says that fluent f is true at step i), $occurs(action, step)$ ($occurs(a, i)$ says that action a occurred at step i), and $fluent(fluent\_type, fluent)$ ($fluent(t, f)$ says that f is a fluent of type t). If l is a literal, $h(l, i)$ will denote $holds(f, i)$ if $l = f$ or $\neg holds(f, i)$ if $l = \neg f$. If p is a set of literals $h(p, i) = \{h(l, i) : l \in p\}$; if e is a set of actions, $occurs(e, i) = \{occurs(a, i) : a \in e\}$.

Definition of $\Pi(\mathcal{D})$

(r1) For every constraint (2), $\Pi(\mathcal{D})$ contains:

$$h(l, I) \leftarrow h(p, I). \tag{5}$$

(r2) $\Pi(\mathcal{D})$ contains the closed world assumption for defined fluents:

$$\begin{aligned} \neg holds(F, I) \leftarrow\ & fluent(defined, F), \\ & \text{not } holds(F, I). \end{aligned} \tag{6}$$

(r3) For every dynamic causal law (1), $\Pi(\mathcal{D})$ contains:

$$\begin{aligned} h(l, I+1) \leftarrow\ & h(p, I), \\ & occurs(a, I). \end{aligned} \tag{7}$$

(r4) For every executability condition (3), $\Pi(\mathcal{D})$ contains:

$$\neg occurs(a_1, I) \text{ v } \ldots \text{ v } \neg occurs(a_k, I) \leftarrow h(p, I). \tag{8}$$

(r5) $\Pi(\mathcal{D})$ contains the Inertia Axiom:

$$\begin{aligned} holds(F, I+1) \leftarrow\ & fluent(inertial, F), \\ & holds(F, I), \\ & \text{not } \neg holds(F, I+1). \end{aligned} \tag{9}$$

$$\begin{aligned} \neg holds(F, I+1) \leftarrow\ & fluent(inertial, F), \\ & \neg holds(F, I), \\ & \text{not } holds(F, I+1). \end{aligned} \tag{10}$$

(r6) and the following rules:

$$fluent(F) \leftarrow fluent(Type, F). \tag{11}$$

$$\leftarrow fluent(F), \text{not } holds(F, I), \text{not } \neg holds(F, I). \tag{12}$$

$$\leftarrow fluent(static, F), holds(F, I), \neg holds(F, I+1). \tag{13}$$

$$\leftarrow fluent(static, F), \neg holds(F, I), holds(F, I+1). \tag{14}$$

(The last four encodings ensure the completeness of states – (11) and (12) – and the proper behavior of static fluents – (13) and (14)). This ends the construction of $\Pi(\mathcal{D})$. Let $\Pi_c(\mathcal{D})$ be a program constructed by rules (r1), (r2), and (r6) above. For any set σ of literals, σ_{nd} denotes the collection of all literals of σ formed by inertial and static fluents. $\Pi_c(\mathcal{D}, \sigma)$ is obtained from $\Pi_c(\mathcal{D}) \cup h(\sigma_{nd}, 0)$ by replacing I by 0.

Definition 1 (State). A set σ of literals is a *state* of $\mathcal{T}(\mathcal{D})$ if $\Pi_c(\mathcal{D}, \sigma)$ has a unique answer set, A, and $\sigma = \{l : h(l, 0) \in A\}$.

Now let σ_0 be a state and e a collection of actions.

$$\Pi(\mathcal{D}, \sigma_0, e) =_{def} \Pi(\mathcal{D}) \cup h(\sigma_0, 0) \cup occurs(e, 0)\ .$$

Definition 2 (Transition). A *transition* $\langle \sigma_0, e, \sigma_1 \rangle$ is in $\mathcal{T}(\mathcal{D})$ iff $\Pi(\mathcal{D}, \sigma_0, e)$ has an answer set A such that $\sigma_1 = \{l : h(l, 1) \in A\}$.

To illustrate the definition we briefly consider

Example 1 (Lin's Briefcase). ([11])
The system description defining this domain consists of: (a) a signature containing the sort name *latch*, the sorted universe $\{l_1, l_2\}$, the action $toggle(latch)$, the inertial fluent $up(latch)$ and the defined fluent *open*, and (b) the following axioms:

$toggle(L)$ **causes** $up(L)$ **if** $\neg up(L)$
$toggle(L)$ **causes** $\neg up(L)$ **if** $up(L)$
$open$ **if** $up(l_1), up(l_2)$.

One can use our definitions to check that the system contains transitions
$\langle\{\neg up(l_1), up(l_2), \neg open\}, toggle(l_1), \{up(l_1), up(l_2), open\}\rangle$,
$\langle\{up(l_1), up(l_2), open\}, toggle(l_1), \{\neg up(l_1), up(l_2), \neg open\}\rangle$, etc.

Note that a set $\{\neg up(l_1), up(l_2), open\}$ is not a state of our system.

System descriptions of $\mathcal{AL}_d$ not containing defined fluents are identical to those of $\mathcal{AL}$. For such descriptions our semantics is equivalent to that of [12], [13]. (To the best of our knowledge, [12] is the first work which uses ASP to describe the semantics of action languages. The definition from [1],[13] is based on rather different ideas.) Note that the semantics of $\mathcal{AL}_d$ is non-monotonic and hence, in principle, the addition of a new definition could substantially change the diagram of $\mathcal{D}$. The following proposition shows that this is not the case. To make it precise we will need the following definition from [14].

Definition 3 (Residue). Let $\mathcal{D}$ and $\mathcal{D}'$ be system descriptions of $\mathcal{AL}_d$ such that the signature of $\mathcal{D}$ is part of the signature of $\mathcal{D}'$. $\mathcal{D}$ is a *residue* of $\mathcal{D}'$ if restricting the states and actions of $\mathcal{T}(\mathcal{D}')$ to the signature of $\mathcal{D}$ establishes an isomorphism between $\mathcal{T}(\mathcal{D})$ and $\mathcal{T}(\mathcal{D}')$.

Proposition 1. Let $\mathcal{D}$ be a system description of $\mathcal{AL}_d$ with signature Σ, $f \notin \Sigma$ be a new symbol for a defined fluent, and $\mathcal{D}'$ be the result of adding to $\mathcal{D}$ the definition of f. Then $\mathcal{D}$ is a residue of $\mathcal{D}'$.

3 Syntax of $\mathcal{ALM}$

A *system description*, $\mathcal{D}$, of $\mathcal{ALM}$ consists of the *system's declarations* (a non-empty set of *modules*) followed by the *system's structure.*

system description *name*
  declarations of *name*
    $[module]^+$
  structure of *name*
    structure description

A *module* can be viewed as a collection of declarations of *sort*, *fluent* and *action* classes of the system, i.e.

module *name*
  sort declarations
  fluent declarations
  action declarations

If the system declaration contains only one module then the first line above can be omitted. In the next two subsections we will define the declarations and the structure of a system description $\mathcal{D}$.

3.1 Declarations of $\mathcal{D}$

(1) A *sort declaration* of $\mathcal{ALM}$ is of the form

$$s_1 \ : \ s_2$$

where s_1 is a sort name and s_2 is either a sort name or the keyword **sort**[1]. In the latter case the statement simply declares a new sort s_1. In the former, s_1 is declared as a subsort of sort s_2.

The sort declaration section of a module is of the form

sort declarations
  $[sort\ declaration]^+$

(2) A *fluent declaration* of $\mathcal{ALM}$ is of the form

$f(s_1, \ldots, s_k) \ : \ type$ **fluent**
  axioms
    $[state\ constraint\ .]^+$
end of f

where f is a fluent name, $s_1, \ldots, s_k$ is a list of sort names, and *type* is one of the following keywords: **static**, **inertial**, **defined**. If the list of sort names is empty we omit the parentheses and simply write f. The remaining part – consisting of the keyword **axioms** followed by a non-empty list of state constraints of $\mathcal{AL}_d$ and the line starting with the keywords **end of** – is optional and can be omitted. The statement declares the fluent f with parameters from $s_1, \ldots, s_k$ respectively as static, inertial, or defined.

The fluent declaration section of a module is of the form

fluent declarations
  $[fluent\ declaration]^+$

(3) An *action declaration* of $\mathcal{ALM}$ is of the form

$a_1 \ : \ a_2$
  attributes
    $[attr \ : \ sort]^+$
  axioms
    $[law\ .]^+$
end of a_1

where a_1 is an action name, a_2 is an action name or the keyword **action**, *attr* is an identifier used to name an attribute of the action, and *law* is a dynamic causal law or an executability condition similar to the ones of $\mathcal{AL}_d$ [2]. If $a_2 =$ **action**,

[1] Syntactically, names are defined as identifiers starting with a lower case letter.
[2] Due to space limitations, we only allow executability conditions of $\mathcal{ALM}$ for single actions, i.e. statements of the form **impossible** a_1 **if** p.

the first statement declares a_1 to be a new action class. If a_2 is an action name then the statement declares a_1 as a special case of the action class a_2. The two remaining sections of the declaration contain the names of attributes of this action, and causal laws and executability conditions for actions from this class. Both the attribute and the axiom part of the declaration are optional and can be omitted. With respect to axioms, the difference between $\mathcal{ALM}$ and $\mathcal{AL}_d$ is that in $\mathcal{AL}_d$ actions are understood as action instances while here they are viewed as action *classes*. Also, in $\mathcal{ALM}$ in addition to literals, the bodies of these laws can contain attribute atoms: expressions of the form $attr = c$, where $attr$ is the name of an attribute of the action and c is an element of the corresponding sort. The action declaration section of a module is of the form

$$\begin{array}{l}\textbf{action declarations}\\ \quad [action\ declaration]^+\end{array}$$

The set of sort, fluent and action declarations from the modules of the system description $\mathcal{D}$ will be called the *declaration* of $\mathcal{D}$ and denoted by $decl(\mathcal{D})$. In order to be "well-defined" the declaration of a system description $\mathcal{D}$ should satisfy certain natural conditions designed to avoid circular declarations and other unintuitive constructs. To define these conditions we need the following notation and terminology:
Sort declarations of $decl(\mathcal{D})$ define a directed graph $S(\mathcal{D})$ such that $\langle sort_2, sort_1\rangle \in S(\mathcal{D})$ iff $sort_1 : sort_2 \in \mathcal{D}$. Similarly, the graph $A(\mathcal{D})$ is defined by action declarations from $decl(\mathcal{D})$. We refer to them as the *sort* and *action hierarchies* of $\mathcal{D}$.

Definition 4. The declaration, $decl(\mathcal{D})$, of a system description $\mathcal{D}$ is called *well-formed* if

1. The sort and action hierarchies of $\mathcal{D}$ are trees with roots **sort** and **action** respectively.
2. If $decl(\mathcal{D})$ contains the declarations of $f(s_1, \ldots, s_k)$ and $f(s'_1, \ldots, s'_k)$ then $s_i = s'_i$ for every $1 \leq i \leq k$.
3. If $decl(\mathcal{D})$ contains the declaration of action a with attributes $attr_1 : s_1$, $\ldots$, $attr_k : s_k$ and the declaration of action a with attributes $attr'_1 : s'_1$, $\ldots$, $attr'_m : s'_m$ then $k = m$, and $attr_i = attr'_i$ and $s_i = s'_i$ for every $1 \leq i \leq k$.

From now on we only consider system descriptions with well-formed declarations.

3.2 Structure of $\mathcal{D}$

The structure of a system description $\mathcal{D}$ defines an interpretation of the sorts, fluents, and actions declared in the system's declaration. It consists of the definitions of the sorts and actions of $\mathcal{D}$, and truth assignments for the statics of $\mathcal{D}$. The sorts are defined as follows:

$$\begin{array}{l}\textbf{sorts}\\ \quad [constants \in\ s]^+\end{array}$$

where *constants* is a non-empty list of identifiers not occurring in the declarations of $\mathcal{D}$ and s is a sort name. We will refer to them as *objects* of $\mathcal{D}$. The definition of the sorts is followed by the definition of actions:

actions
 [*instance description*]$^+$

where an *instance description* is defined as follows:

instance $a_1(t_1, \ldots, t_k)$ **where** $cond\ :\ a_2$
 $attr_1 := t_1$
 $\ldots$
 $attr_k := t_k$

where $attr_1, \ldots, attr_k$ are attributes of an action class a_2 or of an ancestor of a_2 in $A(\mathcal{D})$, t's are objects of $\mathcal{D}$ or variables – identifiers starting with a capital letter –, and *cond* is a set of static literals. An instance description without variables will be called an *action instance*. An instance description containing variables will be referred to as an *action schema*, and viewed as a shorthand for the set of action instances, $a_1(c_1, \ldots, c_k)$, obtained from the schema by replacing the variables $V_1, \ldots, V_k$ by their possibles values $c_1, \ldots, c_k$. We say that an *action instance* $a_1(c_1, \ldots, c_k)$ *belongs to the action class* a_2 *and to any action class which is an ancestor of* a_2 *in* $A(\mathcal{D})$. Finally, we define statics as:

statics
 [*state constraint* .]$^+$

where the head of the state constraint is an expression of the form $f(c_1, \ldots, c_k)$ (where f is a static fluent and $c_1, \ldots, c_k$ are properly sorted elements of the universe of $\mathcal{D}$), and the body of the state constraint is a collection of similar expressions. As usual, if the list is empty the keyword **statics** should be omitted.

Example 2. [Basic Travel]
Let us now consider an example of a system description of $\mathcal{ALM}$.

system description *basic_travel*

 declarations of *basic_travel*

  module *basic_geometry*

   sort declarations

    areas : **sort**

   fluent declarations

    $within(areas, areas)$: **static fluent**
     axioms
      $within(A_1, A_2)$ **if** $within(A_1, A), \ within(A, A_2)$.
      $\neg within(A_2, A_1)$ **if** $within(A_1, A_2)$.
      $\neg within(A_1, A_2)$ **if** $disjoint(A_1, A_2)$.

```
        end of $within$

        $disjoint(areas, areas)$ : static fluent
          axioms
              $disjoint(A_2, A_1)$ if $disjoint(A_1, A_2)$.
              $disjoint(A_1, A_2)$ if $within(A_1, A_3), disjoint(A_2, A_3)$.
            $\neg disjoint(A, A)$.
        end of $disjoint$

  module $move\_between\_areas$

    sort declarations

        $things$ : sort
        $movers$ : $things$
        $areas$ : sort

    fluent declarations

        $loc\_in(things, areas)$ : inertial fluent
          axioms
            $loc\_in(T, A_2)$ if $within(A_1, A_2)$, $loc\_in(T, A_1)$.
            $\neg loc\_in(T, A_2)$ if $disjoint(A_1, A_2)$, $loc\_in(T, A_1)$.
        end of $loc\_in$

    action declarations

        $move$ : action
          attributes
            $actor$ : $movers$
            $origin, dest$ : $areas$
          axioms
            $move$ causes $loc\_in(O, A)$ if $actor = O$, $dest = A$.
            impossible $move$ if $actor = O$, $origin = A$, $\neg loc\_in(O, A)$.
            impossible $move$ if $origin = A_1$, $dest = A_2$, $\neg disjoint(A_1, A_2)$.
        end of $move$

structure of $basic\_travel$

  sorts
      $michael, john \in movers$
      $london, paris, rome \in areas$

  actions
      instance $move(O, A_1, A_2)$ where $A_1 \neq A_2$ : $move$
        $actor := O$
        $origin := A_1$
        $dest := A_2$

  statics
      $disjoint(london, paris)$. $disjoint(paris, rome)$. $disjoint(rome, london)$.
```

4 Semantics of $\mathcal{ALM}$

The semantics of a system description $\mathcal{D}$ of $\mathcal{ALM}$ is defined by mapping $\mathcal{D}$ into the system description $\tau(\mathcal{D})$ of $\mathcal{AL}_d$.

1. The signature, Σ, of $\tau(\mathcal{D})$:
The sort names of Σ are those declared in $decl(\mathcal{D})$. The sorted universe of Σ is given by the sort definitions from $\mathcal{D}$'s structure. We assume the domain closure assumption [15], i.e. the sorts of Σ will have no other elements except those specified in their definitions. An expression $f(c_1, \ldots, c_k)$ is a fluent name of Σ if $s_1, \ldots, s_k$ are the sorts of the parameters of f in the declaration of f from $decl(\mathcal{D})$, and for every i, $c_i \in s_i$. The set of action names of Σ is the set of all action instances defined by the structure of $\mathcal{D}$.

2. Axioms of $\tau(\mathcal{D})$:

(i) The state constraints of $\tau(\mathcal{D})$ are the result of grounding the variables of state constraints from $decl(\mathcal{D})$ and of static definitions from the structure of $\mathcal{D}$ by their possible values from the sorted universe of Σ. Already grounded static definitions from the structure of $\mathcal{D}$ are also state constraints of $\tau(\mathcal{D})$.

(ii) To define dynamic causal laws and executability conditions of $\tau(\mathcal{D})$ we do the following: For every action instance a_i of Σ and every action class a such that a_i belongs to a do:

For every causal law and executability condition L of a:

(a) Construct the expression obtained by replacing occurrences of a in L by a_i. For instance, the result of replacing *move* by $move(john, london, paris)$ in the dynamic causal law for the action class *move* will be:

$$move(john, london, paris) \;\; \textbf{causes} \;\; loc\_in(O, A) \;\; \textbf{if} \;\; \begin{array}{l} actor = O, \\ dest = A. \end{array}$$

(b) Ground all the remaining variables in the resulting expressions by properly sorted constants of the universe of $\mathcal{D}$.
The above axiom will turn into the set containing:

$$\begin{array}{llll} move(john, london, paris) & \textbf{causes} & loc\_in(john, paris) & \textbf{if} \;\; actor = john, \\ & & & \quad\;\; dest = paris. \\ move(john, london, paris) & \textbf{causes} & loc\_in(michael, london) & \textbf{if} \;\; actor = michael, \\ & & & \quad\;\; dest = london. \end{array}$$

etc.

(c) Remove the axioms containg atoms of the form $attr = y$ where y is not the value assigned to $attr$ in the definition of instance a_i. Remove atoms of the form $attr = y$ from the remaining axioms.

This transformation turns the first axiom above into:

$$move(john, london, paris) \;\; \textbf{causes} \;\; loc\_in(john, paris).$$

and eliminates the second axiom.

It is not difficult to check that the resulting expressions are causal laws and executability conditions of $\mathcal{AL}_d$ and hence $\tau(\mathcal{D})$ is a system description of $\mathcal{AL}_d$.

5 Representing Knowledge in $\mathcal{ALM}$

In this section we illustrate the methodology of representing knowledge in $\mathcal{ALM}$ by way of several examples.

5.1 Actions as Special Cases

In the introduction we mentioned the action *carry*, defined as "to move while supporting". Let us now declare a new module containing such an action. The example will illustrate the use of modules for the elaboration of an agent's knowledge, and the declaration of an action as a special case of another action.

Example 3. [Carry]
We expand the system description *basic_travel* by a new module, *carrying_things*.

```
module carrying_things
  sort declarations
    areas : sort
    things : sort
    movers : things
    carriables : things
  fluent declarations
    holding(things, things) : inertial fluent

    is_held(things) : defined fluent
      axioms
        is_held(O) if holding(O_1, O).
    end of is_held

    loc_in(things, areas) : inertial fluent
      axioms
        loc_in(T, A) ≡ loc_in(O, A) if holding(O, T).
    end of loc_in
  action declarations
    move : action
      attributes
        actor : movers
        origin, dest : areas
      axioms
        impossible move if actor = O, is_held(O).
    end of move
```

```
$carry$ : $move$
  **attributes**
    $carried\_thing : carriables$
  **axioms**
     **impossible** $carry$ **if** $actor = O,\ carried\_thing = T,\ \neg holding(O, T)$.
**end of** $carry$

$grip$ : **action**
  **attributes**
    $actor : movers$
    $patient : things$
  **axioms**
    $grip$ **causes** $holding(C, T)$ **if** $actor = C,\ patient = T$.
    **impossible** $grip$ **if** $actor = C,\ patient = T,\ holding(C, T)$.
**end of** $grip$
```

Similarly for action *release*.

Let us add this module to the declarations of *basic_travel* and update the structure of *basic_travel* by the definition of sort *carriables*:

$$suitcase \in carriables$$

and a new action

```
**instance** $carry(O, T, A)$ : $carry$
  $actor := O$
  $carried\_thing := T$
  $dest := A$
```

It is not difficult to check that, according to our semantics, the signature of $\tau(travel)$ of the new system description *travel* will be obtained from the signature of $\tau(basic\_travel)$ by adding the new sort, $carriables = \{suitcase\}$, new fluents like $holding(john, suitcase)$, $is\_held(suitcase)$ *etc.*, and new actions like $carry(john, suitcase, london)$, $carry(john, suitcase, paris)$, etc.

In addition, the old system description will be expanded by axioms:

$$carry(john, suitcase, london) \ \textbf{causes} \ loc\_in(john, london)$$

$$loc\_in(suitcase, london) \equiv loc\_in(john, london) \ \textbf{if} \ holding(john, suitcase)$$

Using Proposition 1 it is not difficult to show that the diagram of *travel* is a conservative extension of that for *basic_travel*.

5.2 Library Modules

The modules from the declaration part of *travel* are rather general and can be viewed as axioms describing our commonsense knowledge about motion. Obviously, such axioms can be used for problem solving in many different domains. It is therefore reasonable to put them in a *library* of commonsense knowlege.

A *library module* can be defined simply as a collection of modules available for public use. Such modules can be imported from the library and inserted in the declaration part of a system description that a programmer is trying to build. To illustrate the use of this library let us assume that all the declarations from *travel* are stored in a library module *motion*, and show how this module can be used to solve the following classical KR problem.

Example 4. [Monkey and Banana]
A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. On the floor of the room stands a box. How can the monkey get the bananas? The monkey is expected to take hold of the box, push it under the banana, climb on the box's top, and grasp the banana.

We are interested in finding a reasonably general and elaboration tolerant declarative solution to this problem. The first step will be identifying sorts of objects relevant to the domain. Clearly the domain contains *things* and *areas*. The things move or are carried from one place to another, climbed on, or grasped. This suggests the use of the library module *motion* containing commonsense axiomatization of such actions. We start with the following:

```
system description  monkey_and_banana

  declarations of  monkey_and_banana

    import motion from commonsense_library
```

An $\mathcal{ALM}$ compiler will simply copy all the declarations from the library module *motion* into our system description. Next we will have:

```
    module main

    % The module will contain specific information about the problem domain.

      sort declarations

        things : sort                 boxes : carriables
        movers : things               bananas : things
        monkeys : movers              areas : sort
        carriables : things           places : areas

      fluent declarations

        under(places, things)  :  static fluent

        is_top(areas, things)  :  static fluent

        can_reach(movers, things)  :  defined fluent
          axioms
            can_reach(M, Box)  if  monkeys(M),
                                   boxes(Box),
                                   loc_in(M, L),
                                   loc_in(Box, L).
        $can\_reach(M, Banana)$ **if** $monkeys(M),$
                                       $bananas(Banana),$
                                       $boxes(Box),$
                                       $loc\_in(M, L_1),$
                                       $is\_top(L_1, Box),$
                                       $loc\_in(Box, L),$
                                       $under(L, Banana).$
      **end of** $can\_reach$
    **action declarations**
      $grip$ : **action**
        **attributes**
          $actor : movers$
          $patient : things$
        **axioms**
          **impossible** $grip$ **if** $actor = C,\ patient = T,\ \neg can\_reach(C, T).$
      **end of** $grip$
  **structure of** $monkey\_and\_banana$
    **sorts**
      $m \in monkeys$
      $b \in bananas$
      $box \in boxes$
      $floor, ceiling \in areas$
      $l_1, l_2, l_3, l_4 \in places$
    **actions**
      **instance** $move(m, L)$ **where** $places(L)$ : $move$
        $actor := m$
        $dest := L$
      **instance** $carry(m, box, L)$ **where** $places(L)$ : $carry$
        $actor := m$
        $carried\_thing := box$
        $dest := L$
      **instance** $grip(m, O)$ **where** $O \neq m$ : $grip$
        $actor := m$
        $patient := O$
    **statics**
      $disjoint(L_1, L_2)$ **if** $places(L_1), places(L_2), L_1 \neq L_2.$
      $disjoint(floor, ceiling).$
      $within(L, floor)$ **if** $places(L), \neg is\_top(L, box).$
      $under(l_1, b).$      $is\_top(l_4, box).$
```

One can check that the system description defines a correct transition diagram of the problem. Standard ASP planning techniques can be used together with the ASP translation of the description to solve the problem.

6 Conclusions

In this paper we introduced a modular extension, $\mathcal{ALM}$, of action language $\mathcal{AL}$. $\mathcal{ALM}$ allows definitions of fluents and actions in terms of already defined fluents and actions. System descriptions of the language are divided into a general uninterpreted theory and its domain dependent interpretation. We believe that this facilitates the reuse of knowledge and the organization of libraries. We are currently working on proving some mathematical properties of $\mathcal{ALM}$ and implementing the translation of its theories into logic programs. Finally, we would like to thank V. Lifschitz for useful discussions on the subject of this paper.

References

1. Turner, H.: Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. Journal of Logic Programming 31(1-3), 245–298 (1997)
2. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Workshop on Logic-Based Artificial Intelligence, pp. 257–279. Kluwer Academic Publishers, Norwell (2000)
3. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (2003)
4. Lifschitz, V., Ren, W.: A Modular Action Description Language. In: Proceedings of AAAI-06, pp. 853-859. AAAI Press (2006)
5. Erdoğan, S.T., Lifschitz, V.: Actions as Special Cases. In: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning, pp. 377–387 (2006)
6. Giunchiglia, E., Lifschitz, V.: An Action Language Based on Causal Explanation: Preliminary Report. In: Proceedings of AAAI-98, pp. 623–630. AAAI Press (1998)
7. Hayes, P.J., McCarthy, J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press, Edinburgh (1969)
8. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–386 (1991)
9. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic Causal Theories. Artificial Intelligence 153, 105–140 (2004)
10. Gustafsson, J., Kvarnström, J.: Elaboration Tolerance Through Object-Orientation. Artificial Intelligence 153, 239–285 (2004)
11. Lin, F.: Embracing Causality in Specifying the Indirect Effects of Actions. In: Proceedings of IJCAI-95, pp. 1985–1993. Morgan Kaufmann (1995)
12. Baral, C., Lobo, J.: Defeasible Specifications in Action Theories. In: Proceedings of IJCAI-97, pp. 1441–1446. Morgan Kaufmann Publishers (1997)
13. McCain, N., Turner, H.: A Causal Theory of Ramifications and Qualifications. Artificial Intelligence 32, 57–95 (1995)
14. Erdoğan, S.T.: A Library of General-Purpose Action Descriptions. PhD thesis, The University of Texas at Austin (2008)
15. Reiter, T.: On Closed World Data Bases. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp.119–140. Plenum Press, New York (1978)