Yet Another Modular Action Language

Michael Gelfond and Daniela Inclezan

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

Abstract. The paper presents the syntax and semantics of an action language $\mathcal{ALM}$. The language is used for 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 new language leads to more elegant and concise representations and facilitates creation of libraries of knowledge modules. The methodology of representing knowledge in $\mathcal{ALM}$ is illustrated by a number of examples.

Key words: knowledge representation, reasoning about actions, action languages

1 Introduction

The paper addresses the problem of representing knowledge about dynamic systems. We limit ourselves to 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. A transition $\langle \sigma, a, \sigma' \rangle$ belongs to such a diagram, $\mathcal{T}$, iff σ' may be a state resulting from execution of action a in state σ. The diagram $\mathcal{T}$ contains all physically possible trajectories of the system, and may often be rather large and complex. The problem of finding concise and mathematically accurate descriptions of such transition diagrams is not trivial and has been a subject of research for over 30 years. Its solution requires a good understanding of the nature of causal effects of actions in the presence of complex dependencies between relations of the domain. Attempts to solve this problem led, among other things, to the development of theory of action languages – logical tools for specifying such diagrams. Even though most action languages have a lot in common, they often differ significantly by the underlining assumptions incorporated in their semantics. For instance, the semantics of an action language $\mathcal{AL}$ [**?**], [**?**] incorporates the *inertia axiom* [**?**] 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 connections between $\mathcal{AL}$ and ASP.[1] The semantics of action language $\mathcal{C}$ [**?**] is based on a different assumption – the so

[1] ASP is the language of logic programs under the Answer Set/Stable Models semantics [**?**], which provides a powerful tool for default representation and reasoning.

called *causality principle* – which says that "*Everything true in the world must be caused*". Our language of choice in this paper is $\mathcal{AL}$ but we hope that people with different philosophical or aesthetic inclinations can adopt some of our ideas to their favorite language.

Until now, most of the work on action languages concentrated on finding the ways to *specify direct and indirect effects and executability conditions of actions*. The *problem of describing objects of dynamic domains, including actions and fluents, remained largely unaddressed*. (For rare exceptions see [**?**], [**?**]). Normally such objects were simply denoted by constants or terms. Even though for simple domains such a practice is acceptable it does not scale to richer domains. For instance representation of actions by simple terms does not provide a programmer with means for defining actions as special cases of other, previously defined, actions. This is however a standard practice (e.g. Merriam-Webster Dictionary defines action *carry* as "to move while supporting", i.e. defines *carry* as a special case of *move*). Similar problems exist for definitions of fluents and other objects. Our new language $\mathcal{ALM}$ addresses this problem. Instead of naming objects by simple terms, it expands $\mathcal{AL}$ by allowing to *define objects as instances of classes of objects*. The idea is of course similar to that used for a long time in frame-based knowledge representation languages and in object-oriented programming languages. However in the context of action languages it takes on some rather distinctive features.

The design of $\mathcal{ALM}$ has been guided by the following goals: we want the language to be as simple as possible, but not simpler; it should have precisely defined syntax and semantics, and it should have a modular structure to allow reuse of knowledge and information hiding. We believe that the latter will facilitate creation of libraries of action descriptions and may lead to more efficient implementations of a number of reasoning tasks.

This paper consists of two parts. First we define the syntax and semantics of a simple extension of $\mathcal{AL}$ by so called defined fluents. The resulting language, $\mathcal{AL}_d$ will then be expanded to $\mathcal{ALM}$, which has a modular structure and allows definition of its entities (including actions) as special cases of more general ones.

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

In this section we expand $\mathcal{AL}$ by means for defining new relations in terms of other, previously defined, ones. This will improve flexibility of the language and allow for more elegant and concise representation of dynamic systems.

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*, *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*, and *defined*. Statics are fluents whose truth values cannot be changed by actions. Inertial fluents can be changed by actions and are subject to the law of inertia. Defined fluents are those which are defined in terms of other fluents. They can only be changed as an indirect effect of actions. 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 also assume that for every sort s and constant c of this sort the signature contains a static, $s(c)$.

Direct causal effects of elementary 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 a fluent literal formed by an inertial fluent and p is a collection of arbitrary fluent literals. (**??**) 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 defined atom or an inertial or static literal. If l is static then p must consists of static literals. Otherwise literals of p are arbitrary. (**??**) says that every state satisfying p must satisfy l. *Executability conditions* of $\mathcal{AL}_d$ are statements of the form:

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

The statement says that actions $a_1, \ldots, a_k$ cannot be executed together in any state which satisfies p. We will often 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 often referred to as the *definition of* f. Finally, an expression of the form

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

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

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

Even though syntactically axioms of $\mathcal{AL}_d$ can not contain variables in practice variables are allowed. An axiom with variables is understood as a shorthand for the set of all its ground instantiations.

The above definitions differ from those of $\mathcal{AL}$ only by the addition of defined fluents and a slightly more careful treatment of statics.

2.2 Semantics of $\mathcal{AL}_d$

A system description $\mathcal{D}$ serves as a specification of the transition diagram $\mathcal{T}(\mathcal{D})$ defining all possible trajectories of the corresponding dynamic system. Therefore, to define the semantics of $\mathcal{AL}_d$, we have to define the states and legal transitions of $\mathcal{T}(\mathcal{D})$ for every $\mathcal{D}$ 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 $\neg f \in \sigma$. By σ_{nd} we denote the collection of all literals of σ formed by inertial and static fluents. 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 answer set of a logic program.* We'll construct a program $\Pi(\mathcal{D})$ consisting of logic programming encodings of statement from $\mathcal{D}$. Answer sets of the union of $\Pi(\mathcal{D})$ with encodings of a state σ_0 and action a will determine the states which the system can move into after the execution of a in σ_0.

The signature of $\Pi(\mathcal{D})$ will contain names from the signature of $\mathcal{D}$, two new sorts: *steps* with two constants, 0 and 1, and *fluent_type* with constants *inertial*, *static*, and *defined*, and 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 $fluent(fluent\_type, fluents)$ ($fluent(t, f)$ says that f is a fluent of type t). If l is a literal formed by a non-static fluent, $h(l, i)$ will denote $holds(f, i)$ if $l = f$ or $\neg holds(f, i)$ if $l = \neg f$. Otherwise $h(l, i)$ is simply l. If p is a set of literals $h(p, i) = \{h(l, i) : l \in p\}$.

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

(r1) For every constraint (**??**) $\Pi(\mathcal{D})$ 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 (**??**), $\Pi(\mathcal{D})$ contains

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

(r4) For every executability condition (**??**) $\Pi(\mathcal{D})$ 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 constraints

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

$$\begin{array}{l} \leftarrow fluent(F), \\ \quad \text{not } holds(F, I), \\ \quad \text{not } \neg holds(F, I). \end{array} \tag{12}$$

(The last two rules ensure completeness of states). This ends the construction of $\Pi(\mathcal{D})$. Let $\Pi_c(\mathcal{D})$ be a program constructed by rules (r1) and (r2) above.

Definition 1 (State). A complete and consistent set σ of literals is a *state* of $\mathcal{T}(\mathcal{D})$ if $l \in \sigma$ iff $h(l, 0)$ belongs to the answer set of the program obtained from $\Pi_c(\mathcal{D}) \cup h(\sigma_{nd}, 0)$ by replacing I by 0.

Now let

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

Definition 2 (Transition). A *transition* $\langle \sigma_0, a, \sigma_1 \rangle$ is in $\mathcal{T}(\mathcal{D})$ iff $\Pi(\mathcal{D}, \sigma_0, a)$ 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). ([**?**])
System description defining this domain consists of: (a) a signature with sort name $latch$, sorted universe $\{l_1, l_2\}$, action $toggle(latch)$, inertial fluents $up(latch)$ and defined fluent $open$, and (b) 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 check, for instance, 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.

For the language not containing defined fluents this definition is equivalent to that given in [**?**,**?**]. ([**?**] is the first work which uses ASP to describe the semantics of action languages. The definition from [**?**],[**?**] is based on rather different ideas.) Note that the semantics of $\mathcal{AL}_d$ is non-monotonic and hence, in principle, 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 this precise we will need the following definition from [**?**].

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 events of $\mathcal{T}(\mathcal{D}')$ to the signature of $\mathcal{D}$ establishes an isomorphism between $\mathcal{T}(\mathcal{D})$ and $\mathcal{T}(\mathcal{D}')$.

The following proposition shows that adding to $\mathcal{D}$ a definition of a new defined fluent yields a "conservative extension" of $\mathcal{D}$.

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 $\mathcal{D}$ a definition of f in terms of Σ. Then $\mathcal{D}$ is a residue of $\mathcal{D}'$.

3 The $\mathcal{ALM}$ Primer

In this section we give a semi-formal description of basic ideas and constructs of $\mathcal{ALM}$, and illustrate these ideas and the use of our language by way of examples. A *system description* of $\mathcal{ALM}$ consists of a *collection of modules* followed by a definition of a *dynamic system*. A module is a *collection of class declarations* organized as a simple hierarchy (i.e a forest of trees). Class declarations define the structures of sorts, fluents, and actions of a domain, and their properties. The definition of a dynamic system specifies the system's sorted universe and its collection of actions. The *semantics* of system description $\mathcal{D}$ of $\mathcal{ALM}$ will be given by mapping it into the system description $sem(\mathcal{D})$ of $\mathcal{AL}_d$. The details of syntax as well as those of construction of $sem(\mathcal{D})$ will be explained below.

Note that modules and class declarations of $\mathcal{ALM}$ are viewed as purely syntactic uninterpreted objects similar to logical formulas or declarations of records in programming languages. This is rather different from $\mathcal{AL}_d$ where interpretation is considered to be part of the signature. To specify a particular dynamic system in $\mathcal{ALM}$ one will need to give an interpretation of primitive sorts, fluents, and actions declared in its modules. As a result, *modules can be viewed as functions mapping such interpretations into the corresponding dynamic systems*. (This is somewhat analogous to viewing logical formulas as functions from their interpretations into truth values.) In what follows we give several examples of system descriptions of $\mathcal{ALM}$.

3.1 Basic Move

We start with considering domains with two sorts of simple entities: (i) movable physical objects referred to as *things*, (ii) roughly bounded parts of space or surface having some specific characteristic or function referred to as *areas*. Things of a domain will be moving between its areas. The geography will be described by two primitive static relations *within* and *disjoint* between the two areas. We will also have an inertial fluent, $loc\_in(things, areas)$, which holds if "a thing is located within an area". For simplicity we assume that, for every two distinct areas A_1 and A_2, A_1 is located within A_2, or A_2 is located within A_1, or A_1 and A_2 are disjoint.

We start representation of our dynamic system with a simple *Geography* module. (We use "boldface" to indicate keywords of $\mathcal{ALM}$.)

Module *Basic Geography* (*bg*)
class *areas* : **sort**
class $within(areas, areas)$: **static**
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** $disjoint(A_1, A_2)$.
class $disjoint(areas, areas)$: **static**
axioms
$disjoint(A_2, A_1)$ **if** $disjoint(A_1, A_2)$.
$disjoint(A_1, A_2)$ **if** $within(A_1, A_3)$, $disjoint(A_2, A_3)$.
$disjoint(A_1, A_2)$ **if** $\neg within(A_1, A_2)$, $\neg within(A_2, A_1)$, $A_1 \neq A_2$.
$\neg disjoint(A_1, A_2)$ **if** $within(A_1, A_2)$.
$\neg disjoint(A, A)$.

End of *Basic Geography*

Here *(bg)* is an (optional) *abbreviated name* for the module; **sort** and **static** are roots of its class hierarchy. Axioms of fluent classes are basically non-grounded domain constraints of $\mathcal{AL}_d$. This is our first example of a module of $\mathcal{ALM}$. It is *flat*, i.e. all trees in its hierarchy are of length 1. To simplify definition of the semantics of our language we will later show how *every module, $\mathcal{M}$, of $\mathcal{ALM}$ can be translated into an "equivalent" flat module with the set of non-grounded axioms $\tau(\mathcal{M})$.* (Of course $\tau(bg)$ *consists of (non-grounded) axioms from the class declarations of bg.*)

Now we use *Basic Geography* module to define a module containing classes for inertial fluents and actions.

Module *Basic Move* (*bm*)
import *Basic Geography*
class *things* : **sort**
class $loc\_in(things, areas)$: **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** $disjoint(A_1, A_2)$, $loc\_in(T, A_1)$.
class *move* : **action**
attributes
$actor : things$
$origin, dest : areas$
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 *Basic Move*

The **import** statement is simply a shorthand for the class declarations from *Basic Geography*. The module contains two more root classes, **inertial fluent** and **action**. Axioms of action classes are similar to dynamic causal laws and executability conditions of $\mathcal{AL}_d$ whose action is the name of the class. The only difference is that they may contain *attribute atoms* formed by attributes of $\mathcal{ALM}$ actions (in our case *actor*, *dest*, and *origin*). The *set of axioms, $\tau(bm)$, defined by this module, is the union of $\tau(bg)$ and the axioms from class declarations of bm.*

3.2 Fluents and Actions as Special Cases

The following constructs are used to define new sort, fluent and action classes as special cases of other, previously defined classes:

(i) **class** $sort_1 \ : \ sort_2$
(ii) **class** $fluent\_class\_name_1(sort_1, \ldots, sort_n) \ : \ fluent\_class\_name_2$
(iii) **class** $action\_class\_name_1 \ : \ action\_class\_name_2$.

We illustrate the semantics of these constructs using the following example:

Module *Travel between Cities* (*tbc*)
class $cities, countries \ : \ areas$
class $tbc.disjoint(areas, areas) \ : \ disjoint$
axioms
$tbc.disjoint(A_1, A_2)$ **if** $cities(A_1), cities(A_2)$.
$tbc.disjoint(A_1, A_2)$ **if** $countries(A_1), countries(A_2)$.
class $tbc.move \ : \ move$
attributes
$origin, dest : cities$
End of *Travel between Cities*

The new module inherits all the declarations of *Basic Move* and hence all the declarations of *Basic Geography*. It contains two new sorts *cities* and *countries* declared as sub-sorts of *areas*, a new static $tbc.disjoint(areas, areas)$ defined as a special case of previously defined static *disjoint*, and a new action class *tbc.move* defined as a special case of previously defined action class *move*.

The axioms, $\tau(tbc)$, are defined as *the union* of $\tau(bm)$, axioms from class declarations of *tbc*, axioms obtained from axioms of *move* by replacing *move* by *tbc.move* (case (iii)), and the following axioms (cases (i) and (ii)):

$areas(C)$ **if** $cities(C)$.
$areas(C)$ **if** $countries(C)$.
$disjoint(A_1, A_2) \equiv tbc.disjoint(A_1, A_2)$.

The definition of τ can be naturally generalized to modules containing arbitrary declarations of the form (i), (ii) and (iii).

Now we will *use module tbc to define a dynamic system,* $\mathcal{D}_0$, containing trajectories of possible travels of two people, Michael and John, between London, Paris, and Rome.

Dynamic System $\mathcal{D}_0$
import *Travel between Cities*
Definition of Sorted Universe
$michael, john \in things$.
$london, paris, rome \in cities$.
$uk, france, italy \in countries$.
Definition of Actions
instance $move(O, A_1, A_2)$ **where** $A_1 \neq A_2$: $tbc.move$
$actor := O$ $origin := A_1$ $dest := A_2$
Truth Assignment for Statics
$within(london, uk)$. $within(paris, france)$. $within(rome, italy)$.
End of $\mathcal{D}_0$

Definition of sorted universe satisfies domain closure assumption [**?**] – sorts have no other elements except those specified in the definition. Actions are defined by an *instance schema* – the collection of action instances of class *tbc.move* obtained by replacing variables by their possible values (in our case O is replaced by things and A_1 and A_2 by different cities). For instance, the collection contains actions $move(john, london, paris)$, $move(michael, london, rome)$, etc. Note that $move(john, rome, rome)$, $move(john, uk, italy)$ are not well formed syntactic expressions. Here is one more example of a module.

Module *Travel in Two Dimensions* (*ttd*)
import *Basic Move*
class $horizontal, vertical$: $areas$
class $ttd.disjoint(areas, areas)$: $disjoint$
axioms
$ttd.disjoint(A_1, A_2)$ **if** $horizontal(A_1), horizontal(A_2)$.
$ttd.disjoint(A_1, A_2)$ **if** $vertical(A_1), vertical(A_2)$.
class h_move : $move$
attributes
$origin, dest : horizontal$
class v_move : $move$
attributes
$origin, dest : vertical$
End of *Travel in Two Dimensions*

This module can be used to create system descriptions which allow things to move in both horizontal and vertical "coordinate systems".

3.3 Semantics

As mentioned above the semantics of system description $\mathcal{D}$ of $\mathcal{ALM}$ is defined by mapping it into the system description $sem(\mathcal{D})$ of $\mathcal{AL}_d$. This is done by the following construction:

Let $\mathcal{M}_1, \ldots, \mathcal{M}_k$ be modules imported by a system description $\mathcal{D}$.

(1) $sem(\mathcal{D}) := \tau(\mathcal{M}_1) \cup \ldots \cup \tau(\mathcal{M}_k)$
(Obviously in our example $sem(\mathcal{D}_0)$ is $\tau(tbc)$).

(2) For every action class of $sem(\mathcal{D})$ replace occurrences of its name in dynamic causal laws and executability conditions by names of instances of this class.
(For instance, the result of replacing *move* by $move(john, london, paris)$ in the dynamic causal law for action class *move* will be
$move(john, london, paris)$ **causes** $loc\_in(O, A)$ **if** $actor(O)$, $dest(A)$.
Similarly for other instances.)

(3) Ground all the variables in $sem(\mathcal{D})$ by properly sorted constants of $\mathcal{D}$.
($sem(\mathcal{D}_0)$ will now contain grounded state constraints and axioms like
$move(john, london, paris)$ **causes** $loc\_in(john, paris)$ **if** $actor(john)$, $dest(paris)$.
$move(john, london, paris)$ **causes** $loc\_in(michael, london)$ **if** $actor(michael)$, $dest(london)$.
etc.)

(4) Let A be a causal law or executability condition from $sem(\mathcal{D})$ containing action a. If A contains an attribute atom $attr(y)$ such that the definition of a does not assign y as the value of $attr$, remove A from $sem(\mathcal{D})$. Otherwise remove the attribute atoms of A.
(This transformation turns the first law above into
$move(john, london, paris)$ **causes** $loc\_in(john, paris)$
and eliminates the second law.)

Now $sem(\mathcal{D})$ is a system description of $\mathcal{AL}_d$. $\mathcal{T}(\mathcal{D})$ is obtained from $\mathcal{T}(sem(\mathcal{D}))$ by removing all states not satisfying the truth assignments for statics in $\mathcal{D}$.

3.4 One More Example

In the introduction we mentioned action *carry* defined as "move while supporting". Let us now define a new module, containing such an action.

Module *Carry Things* (*ct*)
import *Basic Move*

class $carriers$: $things$
class $supports(carriers, things)$: **inertial fluent**
class $ct.loc\_in(things, areas)$: loc_in
axioms
$loc\_in(C, A) \equiv ct.loc\_in(T, A)$ **if** $supports(C, T)$.
class $carry$: $move$
attributes
$carried\_thing : things$
axioms
impossible $carry$ **if** $actor(O)$,
$carried\_thing(T)$,
$\neg supports(T, O)$.
class $grip$: **action**
attributes
$carrier : carriers$ $carried\_thing : things$
axioms
$grip$ **causes** $supports(C, T)$ **if** $carrier(C)$,
$carried\_thing(T)$.
impossible $grip$ **if** $carrier(C)$,
$carried\_thing(T)$,
$supports(C, T)$.
class $release$: **action**
attributes
$carrier : carriers$ $carried\_thing : things$
axioms
$release$ **causes** $\neg supports(C, T)$ **if** $carrier(C)$,
$carried\_thing(T)$.
impossible $release$ **if** $carrier(C)$,
$carried\_thing(T)$,
$\neg supports(C, T)$.
End of *Carry Things*

Let us now define a dynamic system, $\mathcal{D}_1$, describing the travel of John who may or may not carry his briefcase with him.

Dynamic System $\mathcal{D}_1$
import *Carry Things*
Definition of Sorted Universe
$john \in carriers$
$briefcase \in things$
$london, paris, rome \in areas$
Definition of Actions
instance $move(Actor, Area)$: $move$
$actor := Actor$
$dest := Area$

instance $carry(Actor, Thing, Area)$ **where** $Actor \neq Thing$: $carry$
$actor := Actor$
$dest := Area$
$carried\_thing := Thing$
instance $grip(C, T)$ **where** $C \neq T$: $grip$
$carrier := C$
$carried\_thing := T$
instance $release(C, T)$ **where** $C \neq T$: $release$
$carrier := C$
$carried\_thing := T$

Truth Assignment for Statics
$disjoint(london, paris).$ $disjoint(rome, paris).$ $disjoint(london, rome).$

It is not difficult to check that, according to $\mathcal{T}(\mathcal{D}_1)$, after the execution of a sequence of actions $\langle grip(john, briefcase), carry(john, briefcase, london), release(john, briefcase), move(john, paris)\rangle$ in a suitable initial state, John will be in Paris while his briefcase will stay in London.

4 Related Work

We are aware of only one earlier extension of $\mathcal{AL}$ by defined fluents [**?**], [**?**] but it is different from ours. For instance, in this work an action description with one state constraint, g **if** g, where g is a defined fluent, will have two states, $\{g\}$ and $\{\neg g\}$, while in our approach it will only have one, $\{\neg g\}$. Both approaches however coincide for a large class of action theories, not allowing circular dependencies between fluents. The precise description of this class is not directly related to our main goal and will be given elsewhere. Language $\mathcal{ALM}$ has its roots in modular ASP based knowledge representation language $\mathcal{M}$ briefly discussed in [**?**]. Even though the intuitive ideas of $\mathcal{M}$ are similar to those of $\mathcal{ALM}$, the details are substantially different. In particular the new language is limited to theories of actions and its semantics is both more accurately described and substantially simpler than that of $\mathcal{M}$.

$\mathcal{ALM}$ has also close conceptual relation to modular action language, $\mathcal{MAD}$ [**?**],[**?**]. Its high level goals seem to be rather similar to ours but there are very substantial differences between the two languages. $\mathcal{MAD}$ expands the extension $\mathcal{C}+$ [**?**] of $\mathcal{C}$ and is, therefore, based on *causal logic*. Even though there is a relationship between $\mathcal{C}+$ and ASP it is substantially less straighforward than that of $\mathcal{AL}_d$. Another important difference is that, in $\mathcal{MAD}$, the description of the universe is viewed as part of the module while $\mathcal{ALM}$ maintains a strong separation between the two. In addition, $\mathcal{MAD}$ has a substantially more complex import mechanism, requires declaration of variables, etc. Overall we believe that $\mathcal{ALM}$ is conceptually and technically simpler than $\mathcal{MAD}$ but of course a substantial amount of work is required to prove or disprove this hypothesis. One advantage of $\mathcal{MAD}$ is its ability to deal with non-boolean fluents but we

believe that adding this feature to $\mathcal{AL}_d$ and hence to $\mathcal{ALM}$ is a fairly routine task. Finally we will mention object-oriented language TAL-C [**?**] which allows definitions of classes somewhat similar to those in $\mathcal{ALM}$ and $\mathcal{MAD}$. The goal of TAL-C however seems to be more ambitious than that of these action languages. The language is used to describe and reason about various dynamic scenarios. In $\mathcal{ALM}$, description of a scenario and that of reasoning tasks are not viewed as part of the language.

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. 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)
4. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–386 (1991)
5. Giunchiglia, E., Lifschitz, V.: An Action Language Based on Causal Explanation: Preliminary Report. In: Proceedings of AAAI-98, pp. 623–630. AAAI Press (1998)
6. Lifschitz, V., Ren, W.: A Modular Action Description Language. In: Proceedings of AAAI-06, pp. 853-859. AAAI Press (2006)
7. Gustafsson, J., Kvarnström, J.: Elaboration Tolerance Through Object-Orientation. Artificial Intelligence 153, 239–285 (2004)
8. Lin, F.: Embracing Causality in Specifying the Indirect Effects of Actions. In: Proceedings of IJCAI-95, pp. 1985–1993. Morgan Kaufmann (1995)
9. Baral, C., Lobo, J.: Defeasible Specifications in Action Theories. In: Proceedings of IJCAI-97, pp. 1441–1446. Morgan Kaufmann Publishers (1997)
10. McCain, N., Turner, H.: A Causal Theory of Ramifications and Qualifications. Artificial Intelligence 32, 57–95 (1995)
11. Erdoğan, S.T.: A Library of General-Purpose Action Descriptions. PhD thesis, The University of Texas at Austin (2008)
12. 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)
13. Gelfond, M., Watson, R.: On Methodology of Representing Knowledge in Dynamic Domains. Sci. Comput. Program. 42(1), 87–99 (2002)
14. Watson, R.: Action Languages for Domain Modeling. PhD thesis, University of Texas at El Paso (1999)
15. Gelfond, M.: Going Places - Notes on a Modular Development of Knowledge about Travel. In: AAAI 2006 Spring Symposium Series, pp. 56–66 (2006)
16. 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)
17. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic Causal Theories. Artificial Intelligence 153, 105–140 (2004)