Reasoning about Dynamic Domains in Modular Action Language $\mathcal{ALM}$

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. They represent basic relations in the domain to be modeled, the quitessential effects of actions in the domain. Defined fluents are non-static fluents which are defined in terms of other fluents. They can be changed by actions but only indirectly. Defined fluents are used to deal with the complexity of the domain; they are helpful shorthands for the modeler of the domain.

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:

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

The statement says that actions $a_1, \ldots, a_k$ cannot be executed together in any state which satisfies p. We 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
$\neg f$ **if** $p, \neg g$
g **if** p, f
$\neg g$ **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:

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

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

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

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

(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. Syntactically, names are defined as identifiers starting with a lower case letter. 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** 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 $\mathcal{AL}_d$. If a_2 = **action**, 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. In executability conditions preventing the simultaneous execution of two or more instances of the same action class, the occurrences of the class name will be indexed to distinguish between different instances. For example, if we had an action class *move* and we wanted to say that no two instances of this action class can be executed simultaneously, we would write it as:

impossible $move_1, move_2$

The general form of executability conditions in $\mathcal{ALM}$ is:

$$\textbf{impossible}\ a_{1_1}, \ldots, a_{1_m}, \ldots\ldots, a_{k_1}, \ldots, a_{k_n}\ \textbf{if}\ p \tag{15}$$

The set of possibly indexed names of action classes appearing in an executability condition L of $\mathcal{ALM}$ will be called the *action classes of* L and will be denoted by $aclasses(L)$:

$$aclasses(L) = \{a_{1_1}, \ldots, a_{1_m}, \ldots\ldots, a_{k_1}, \ldots, a_{k_n}\}.$$

For example: $aclasses($ **impossible** $move_1, move_2) = \{move_1, move_2\}$.

If $aclasses(L)$ contains more than one element, then any attribute atom that appears in p and is a common attribute between two or more action classes in $aclasses(L)$ will be indexed with the (possibly indexed) name of the action class it belongs to. For example:

impossible $move_1, move_2$	**if**	$actor_{move_1} = A_1$, $actor_{move_2} = A_2$, $A_1 = A_2$.
impossible $grip, release$	**if**	$actor_{grip} = A_1$, $actor_{release} = A_2$, $A_1 = A_2$.
impossible $move_1, move_2, grip$	**if**	$actor_{move_1} = A_1$, $actor_{move_2} = A_2$, $actor_{grip} = A_3$, $A_1 = A_2$, $A_2 = A_3$.

The action declaration section of a module is of the form

action declarations
$[action\ declaration]^+$

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 directed *acyclic* graphs with sources **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 $attr_1 : s_1$, ..., $attr_k : s_k$ and the declaration of action a with attributes $attr'_1 : s'_1$, ..., $attr'_m : s'_m$ 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:

sorts
[*constants* $\in$ *s*]$^+$

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 $\mathcal{ALM}$.

We consider a domain with three sorts of simple entities: (i) separate and self-contained entities referred to as *things*; (ii) *things* that can move on their own, referred to as *movers*; and (iii) roughly bounded parts of space or surface having some specific characteristic or function, referred to as *areas*. The *movers* of

this domain will be moving between areas of the domain. The geometry will be described by two primitive static relations between two areas: *within* and *disjoint* (an area A_1 can be located within an area A_2; two areas, A_1 and A_2, can be disjoint from each other). We will also have an inertial fluent, *loc_in(things, areas)*, which holds if a thing is located within an area.

system description *basic_travel*

declarations of *basic_travel*

module *basic_geometry*

sort declarations

$areas$: **sort**
% roughly bounded parts of space or surface
% having some specific characteristic or function

fluent declarations

$within(areas, areas)$: **static fluent**
% one area is located within another area

axioms

$within(A_1, A_2)$ **if** $within(A_1, A)$, $within(A, A_2)$.
$\neg within(A_2, A_1)$ **if** $within(A_1, A_2)$, $A_1 \neq A_2$.
$\neg within(A_1, A_2)$ **if** $disjoint(A_1, A_2)$.

end of *within*

$disjoint(areas, areas)$: **static fluent**
% one area is disjoint from another area

axioms

$disjoint(A_2, A_1) \equiv$ $disjoint(A_1, A_2)$.
$disjoint(A_1, A_2)$ **if** $within(A_1, A_3)$, $disjoint(A_2, A_3)$.
$\neg disjoint(A_1, A_2)$ **if** $within(A_1, A_2)$.
$\neg disjoint(A, A)$.

end of *disjoint*

module *move_between_areas*

sort declarations

$things$: **sort**
% separate and self-contained entities
$movers$: $things$
% things that can move on their own
$areas$: **sort**

fluent declarations

$loc\_in(things, areas)$: **inertial fluent**
% one thing is located in an area
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** $actor = O$, $loc\_in(O, A_1)$, $dest = A_2$, $\neg disjoint(A_1, A_2)$.[1]
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
$disjoint(london, paris)$.
$disjoint(paris, rome)$.
$disjoint(rome, london)$.
$\neg within(A, A)$.

The first axiom in the declaration of the action class *move* is to be read as: For every instance m of the action class *move*, if O is the actor of m and A is the destination of m, then the occurrence of m causes O to be located in A.

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 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 dynamic causal law 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:

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

(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:

$$move(john, london, paris) \ \textbf{causes} \ loc\_in(john, paris) \ \textbf{if} \ actor = john, \ dest = paris.$$
$$move(john, london, paris) \ \textbf{causes} \ loc\_in(michael, london) \ \textbf{if} \ actor = michael, \ dest = london.$$

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.

(iii) In order to define the executability conditions of $\tau(\mathcal{D})$ we need to introduce some notation first:

Let $m(aclasses(L))$, where L is an executability condition from $\mathcal{D}$, be a *bijection*[2] from the set $aclasses(L)$ into the set of action instances from the signature Σ of $\tau(\mathcal{D})$. We will call $m(aclasses(L))$ a *mapping of action classes of L into action instances*.

Note that there can be multiple different mappings $m(aclasses(L))$ for an executability condition L. For example, if $decl(basic\_travel)$ contained an executability condition stating that an actor cannot perform two different actions of the type *move* simultaneously[3]:

$$\begin{array}{lll} \textbf{impossible}\ move_1, move_2 & \textbf{if} & actor_{move_1} = A_1, \\ & & actor_{move_2} = A_2, \\ & & A1 = A2. \end{array} \tag{16}$$

then we could have the mappings:

$m_1(move_1) = move(john, paris, london)$,
$m_1(move_2) = move(john, paris, rome)$

$m_2(move_1) = move(michael, paris, london)$,
$m_2(move_2) = move(john, london, rome)$

etc.

We can now define the semantics of executability conditions of $\mathcal{D}$:

For every executability condition L of $\mathcal{D}$ do:

For every mapping $m(aclasses(L))$:

(a) Construct the expression obtained by replacing every action class a of $aclasses(L)$ by $m(a)$.
For example, if $decl(basic\_travel)$ contained the executability condition in (16), then the mapping m_1 above would produce the expression:

$$\begin{array}{l} \textbf{impossible}\ move(john, paris, london), move(john, paris, rome) \\ \quad \textbf{if}\ \ actor_{move_1} = A_1, \\ \qquad actor_{move_2} = A_2, \\ \qquad A1 = A2. \end{array}$$

(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:

$$\begin{array}{l} \textbf{impossible}\ move(john, paris, london), move(john, paris, rome) \\ \quad \textbf{if}\ \ actor_{move_1} = john, \\ \qquad actor_{move_2} = john, \\ \qquad john = john. \end{array}$$

impossible $move(john, paris, london), move(john, paris, rome)$
if $actor_{move_1} = john$,
$actor_{move_2} = michael$,
$john = michael$.
etc.

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

This transformation turns the first axiom above into:

impossible $move(john, paris, london), move(john, paris, rome)$
if $john = john$.
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*
% things which can be carried by movers

fluent declarations

holding(*things*, *things*) : **inertial fluent**
% *holding(X,Y)* if X holds Y in position so as to keep it from falling

$is\_held(things)$: **defined fluent**
% a thing is held in position
axioms
$is\_held(O)$ **if** $holding(O_1, O)$.
end of is_held

$loc\_in(things, areas)$: **inertial fluent**
axioms
$loc\_in(T, A) \equiv loc\_in(O, A)$ **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** $actor = C$,
$patient = T$.
$grip$ **causes** $loc\_in(T, A)$ **if** $actor = C$,
$patient = T$,
$loc\_in(C, A)$.
impossible $grip$ **if** $actor = C$,
$patient = T$,
$holding(C, T)$.
end of $grip$

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

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:

$carry(john, suitcase, london)$ **causes** $loc\_in(john, london)$

$loc\_in(suitcase, london) \equiv loc\_in(john, london)$ **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 The Use of Library Modules for Creating System Descriptions

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, carry 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**
$movers$: $things$
$monkeys$: $movers$
$carriables$: $things$
$boxes$: $carriables$
$bananas$: $things$
$areas$: **sort**
$places$: $areas$

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** $monkeys(M)$,
$boxes(Box)$,
$places(L)$,
$loc\_in(M, L)$,
$loc\_in(Box, L)$.

$can\_reach(M, Banana)$ **if** $monkeys(M)$,
$bananas(Banana)$,
$boxes(Box)$,
$places(L_1)$,
$places(L)$,
$loc\_in(M, L_1)$,
$is\_top(L_1, Box)$,
$loc\_in(Box, L)$,
$under(L, Banana)$.

end of *can_reach*

action declarations

move : **action**
attributes
$actor : movers$
$origin, dest : areas$
axioms
impossible *move* **if** $actor = O$,
$monkeys(O)$,
$dest = L$,
$is\_top(L, Box)$,
$holding(O, Box)$.[4]

end of *move*

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

$m \in monkeys$
$b \in bananas$
$box \in boxes$
$floor, ceiling \in areas$
$l_1, l_2, l_3, l_4, l_5 \in places$

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** $O \neq m$: $grip$
$actor := m$
$patient := O$
instance $release(m, O)$ **where** $O \neq m$: $release$
$actor := m$
$patient := O$

statics

$within(L, floor)$ **if** $places(L)$, $L \neq l_4$, $L \neq l_5$.
$within(l_5, ceiling)$.
$\neg within(A, A)$.
$disjoint(L_1, L_2)$ **if** $places(L_1)$, $places(L_2)$, $L_1 \neq L_2$.
$disjoint(floor, ceiling)$.
$disjoint(l_4, floor)$.
$disjoint(l_4, ceiling)$.
$under(l_1, b)$.
$\neg under(L, b)$ **if** $L \neq l_1$.
$\neg under(L, Thing)$ **if** $Thing \neq b$.
$is\_top(l_4, box)$.
$\neg is\_top(L, box)$ **if** $L \neq l_4$.
$\neg is\_top(L, Thing)$ **if** $Thing \neq box$.

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. For example, if in the initial situation the monkey is in l_2, the box is in l_3 and the banana is in l_5, two possible plans are found. The first plan consists of the sequence of actions:

$occurs(move(m, l_3), 0)$
$occurs(grip(m, box), 1)$
$occurs(carry(m, box, l_1), 2)$
$occurs(release(m, box), 3)$
$occurs(move(m, l_4), 4)$
$occurs(grip(m, b), 5)$

The second plan contains the action $occurs(move(m, l_1), 2)$ instead of the action $occurs(carry(m, box, l_1), 2)$.

6 Examples

6.1 Networks

Networks can be used to represent many real world domains: electrical circuits, the Internet, logical gates, cities connected by roads etc. It would be useful to have a general library module that represents commonsense knowledge about networks.

We can think of networks as being special cases of multigraphs, most commonly: directed multigraphs. Therefore, we will start by creating a library module on directed multigraphs. The objects of this domain are nodes and directed edges. An edge links one node to another. The library module representing this knowledge is the following one:

module *directed_multigraph*

sort declarations

nodes : **sort**
edges : **sort**

fluent declarations

$links(nodes, edges, nodes)$: **inertial fluent**
% $links(N_1, E, N_2)$ if the directed edge
% named E links node N_1 to node N_2

$connected(nodes, nodes)$: **defined fluent**
axioms
$connected(N_1, N_2)$ **if** $links(N_1, E, N_2)$.
$connected(N_1, N_3)$ **if** $connected(N_1, N_2)$, $connected(N_2, N_3)$.
end of *connected*

cyclical : **defined fluent**
% *cyclical* is true if the graph is cyclical
axioms
cyclical **if** $connected(N, N)$.
end of *cyclical*

action declarations

add_link : **action**
attributes
$arc : edges$
$start, end : nodes$
axioms

add_link **causes** $links(N_1, E, N_2)$ **if** $arc = E$, $start = N_1$, $end = N_2$.

impossible *add_link* **if** $arc = E$, $start = N_1$, $end = N_2$, $links(N_1, E, N_2)$.

end of *add_link*

remove_link : **action**

attributes

$arc : edges$

$start, end : nodes$

axioms

remove_link **causes** $\neg links(N_1, E, N_2)$ **if** $arc = E$, $start = N_1$, $end = N_2$.

impossible *remove_link* **if** $arc = E$, $start = N_1$, $end = N_2$, $\neg links(N_1, E, N_2)$.

end of *remove_link*

We can now use this library modules to represent the domain of e-commerce shipments. In this domain, we have things located at different addresses. Addresses are locations, disjoint from one another, where things can be stored. Things can be transported from one address to another. Addresses are connected by roads. There can be different paths from one address to another. A road connecting two addresses can be interrupted. We will ignore the medium of transportation for now, and consider that the things can move by themselves to their destination. In order to represent this domain, we can re-use the *move_between_areas* library module for the transportation part, and substitute the *basic_geometry* module with the new *undirected_multigraph* module.

system description *ecomm_delivery_logistics*

declarations of *ecomm_delivery_logistics*

import *directed_multigraph* **from** *commonsense_library*

import *move_between_areas* **from** *commonsense_library*

module *main*

sort declarations

areas : **sort**

things : **sort**

movers : *things*

nodes : *areas*

edges : **sort**

addresses : *nodes*

fluent declarations

$disjoint(areas, areas)$: **static fluent**
axioms
$disjoint(A_1, A_2)$ **if** $nodes(A_1)$, $nodes(A_2)$, $A_1 \neq A_2$.
end of disjoint

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

action declarations

$shipment$: $move$
axioms
impossible $shipment$ **if** $actor = Item$, $loc\_in(Item, A_1)$, $dest = A_2$, $\neg connected(A_1, A_2)$.
end of $shipment$

structure of $ecomm\_delivery\_logistics$

sorts

$plasmaTV46 \in movers$
$warehouse\_LasVegasNV, addr\_9\_FogSt\_SeattleWA \in addresses$
$i\_84\_W, i\_5\_N \in edges$

actions

instance $shipment(Item, Addr_1, Addr_2)$ **where** $Addr_1 \neq Addr_2$: $shipment$
$actor := Item$
$origin := Addr_1$
$dest := Addr_2$

instance $add\_link(E, A_1, A_2)$: add_link
$arc := E$
$start := A_1$
$end := A_2$

instance $remove\_link(E, A_1, A_2)$: $remove\_link$
$arc := E$
$start := A_1$
$end := A_2$

statics

$\neg within(A, A)$.

If in the initial situation at least one of the two roads, i_84_W and i_5_N, links the address $warehouse\_LasVegasNV$ to $addr\_9\_FogSt\_SeattleWA$, then the shipment can be executed. But if both linking roads are made unaccessible by performing an action of the type *remove_link* (the traffic on the road is obstructed, etc.), then the *shipment* action can no longer be accomplished.

Networks are also characterized by flow. For some networks this flow can be represented using the basic action *move*, as in the previous example. However, in other cases the flow moves automatically through the whole network, as in the case of a signal passing through a digital circuit. A signal is "a detectable physical quantity or impulse (as a voltage, current, or magnetic field strength) by which messages or information can be transmitted." Only *source_nodes*, a special sort of *nodes* can generate signals. The value of the signal on an edge (if the edge is not directly connected to a *source_node*) depends on the node the edge comes out of.

module *flow_network*

sort declarations

nodes : **sort**
edges : **sort**

signals : **sort**
% a detectable physical quantity or impulse
% by which messages or information can be transmitted

source_nodes : *nodes*
% nodes that can apply signals on edges

fluent declarations

$value(edges, signals)$: **inertial fluent**
axioms
$\neg value(E, S_1)$ **if** $value(E, S_2)$, $S_1 \neq S_2$.
end of *value*

$input(edges, nodes)$: **defined fluent**
axioms
$input(E, N_2)$ **if** $links(N_1, E, N_2)$.
end of *input*

$output(edges, nodes)$: **defined fluent**
axioms
$output(E, N_1)$ **if** $links(N_1, E, N_2)$.
end of *output*

action declarations

apply_signal : **action**
  attributes
    source_node : *source_nodes*
    signal : *signals*
  axioms
    apply_signal **causes** *value*(E, S) **if** *source_node* $= N$,
                                                signal $= S$,
                                                output(E, N).
end of *apply_signal*

We can now address the representation of digital circuits. Here, the edges will be the wires connecting different logical gates. The logical gates (*not, and, or*) will be nodes. We will consider three possible signals: 0, 1, and 0.5 (undefined). The following library module represents knowledge about digital circuits:

module *digital_circuits*

sort declarations

nodes : **sort**
edges : **sort**

signals : *rational_numbers*
% a detectable physical quantity or impulse
% by which messages or information can be transmitted

source_nodes : *nodes*
% nodes that can apply signals on edges

gates : *nodes*
not_gates : *gates*
and_gates : *gates*
or_gates : *gates*

fluent declarations

$value(edges, signals)$: **inertial fluent**
axioms
$value(W, 0.5)$ **if** $output(W, G)$,
$all\_input\_values(G, 0.5)$.
$value(W, S)$ **if** $output(W, G)$,
$not\_gates(G)$,
$input(W_0, G)$,
$value(W_0, S_0)$,
$opposite(S, S_0)$.
$value(W, 0)$ **if** $output(W, G)$,
$and\_gates(G)$,
$input(W_0, G)$,
$value(W_0, 0)$.
$value(W, 1)$ **if** $output(W, G)$,
$and\_gates(G)$,
$all\_input\_values(G, 1)$.
$value(W, 1)$ **if** $output(W, G)$,
$or\_gates(G)$,
$input(W_0, G)$,
$value(W_0, 1)$.
$value(W, 0)$ **if** $output(W, G)$,
$or\_gates(G)$,
$all\_input\_values(G, 0)$.
end of *value*

$opposite(signals, signals)$: **static fluent**
axioms
$opposite(S_1, S_2) \equiv opposite(S_2, S_1)$.
end of *opposite*

$all\_input\_values(gates, signals)$: **defined fluent**
axioms
$all\_input\_values(G, S)$ **if** $has\_signals\_on\_inputs(G)$,
$\neg different\_input\_values(G, S)$.
end of *all_input_values*

$has\_signals\_on\_inputs(gates)$: **defined fluent**
axioms
$has\_signals\_on\_inputs(G)$ **if** $input(W, G)$,
$value(W, S)$.
end of *has_signals_on_inputs*

$different\_input\_values(gates, signals)$: **defined fluent**
axioms
$different\_input\_values(G, S_1)$ **if** $input(W, G)$, $value(W, S_2)$, $S_1 \neq S_2$.
end of $different\_input\_values$

$links(nodes, edges, nodes)$: **inertial fluent**
axioms
$\neg links(G_0, W_1, G)$ **if** $input(W_2, G)$, $not\_gates(G)$, $W_1 \neq W_2$.
end of $links$

We will use the library modules *directed_multigraph*, *flow_network*, and *digital_circuits* to represent a domain in which we have one gate of each type (*and*, *or*, *not*), three *source_nodes* and one final node.

system description *digital_circuit_example*

declarations of *digital_circuit_example*

import *directed_multigraph* **from** *commonsense_library*

import *flow_network* **from** *commonsense_library*

import *digital_circuits* **from** *commonsense_library*

structure of *digital_circuit_example*

sorts

$n_1, n_2, n_3 \in source\_nodes$
$n_4 \in nodes$
$not\_gate \in not\_gates$
$and\_gate \in and\_gates$
$or\_gate \in or\_gates$
$w_1, w_2, w_3, w_4, w_5, w_6 \in edges$
$0, 0.5, 1 \in signals$

actions

instance $apply\_signal(N, S)$: $apply\_signal$
$source\_node := N$
$signal := S$

instance $add\_link(E, N_1, N_2)$: add_link
$arc := E$
$start := N_1$
$end := N_2$

instance $remove\_link(E, N_1, N_2)$: $remove\_link$
$arc := E$
$start := N_1$
$end := N_2$

statics

$opposite(0, 1)$.
$opposite(0.5, 0.5)$.
$\neg opposite(0.5, S)$ **if** $S \neq 0.5$.
$\neg opposite(S, S)$ **if** $S \neq 0.5$.

If we attached to the ASP encoding of this system description a history

$holds(links(n_1, w_1, and\_gate), 0)$
$holds(links(n_2, w_2, and\_gate), 0)$
$holds(links(n_3, w_3, not\_gate), 0)$
$holds(links(and\_gate, w_4, or\_gate), 0)$
$holds(links(not\_gate, w_5, or\_gate), 0)$
$holds(links(or\_gate, w_6, n_4), 0)$
$occurs(apply\_signal(n_1, 1), 0)$
$occurs(apply\_signal(n_2, 1), 0)$
$occurs(apply\_signal(n_3, u), 0)$

then, at time step 1, the values of the signals on the wires would be:

$holds(value(w_1, 1), 1)$.
$holds(value(w_2, 1), 1)$.
$holds(value(w_3, u), 1)$.
$holds(value(w_4, 1), 1)$.
$holds(value(w_5, u), 1)$.
$holds(value(w_6, 1), 1)$.

We can now consider another example of a network: an Ethernet network. An Ethernet network connects different devices (computers, hubs, switches etc.) via cable. Computers transmit packets of information from one to another via the cable. A packet has only one source, but it can have multiple destinations (for example, when packets are broadcasted in the network). The signal representing the packet can degradate if coaxial cable is used and the cable is too long. To prevent signal degradation and to allow multiple devices to be connected simultaneously, repeaters such as hubs can be used. A hub broadcasts packets entering any of its ports out on all its other ports. A switch is another device that can be used in an Ethernet network. A switch is smarter than a hub in the sense that it inspects the packet, finds out what its destination is, and only transmits the packet on the relevant cable (the one connected to the destination of the packet). We will represent this simplified description of an Ethernet network as follows:

module *ethernet_network*

sort declarations

$nodes$: **sort**
$edges$: **sort**

$signals$: **sort**
$packets$: $signals$
% blocks of data that are individually sent and delivered
% from one computer to another

$source\_nodes$: $nodes$
$computers$: $source\_nodes$
$hubs$: $nodes$
$switches$: $nodes$
$wires$: $edges$

fluent declarations

$destination(packets, computers)$: **static fluent**

$source(packets, computers)$: **static fluent**
axioms
$\neg source(P, C_1)$ **if** $source(P, C_2)$,
$C_1 \neq C_2$.
end of $source$

$links(nodes, edges, nodes)$: **inertial fluent**
axioms
$links(N_2, E, N_1) \equiv links(N_1, E, N_2)$.
end of $links$

$received(packets)$: **defined fluent**
axioms
$received(P)$ **if** $destination(P, C)$,
$input(W, C)$,
$value(W, P)$.
end of $received$

$leads\_to(edges, nodes)$: **defined fluent**
% $leads\_to(E, N)$ is true if there is a path
% from the endpoint of E to N
axioms
$leads\_to(E, N)$ **if** $input(E, N)$.
$leads\_to(E, N_2)$ **if** $input(E, N_1)$,
$connected(N_1, N_2)$.
end of $leads\_to$

$value(edges, signals)$: **inertial fluent**
axioms
$value(W_1, P)$ **if** $output(W_1, N)$,
$hubs(N)$,
$input(W_2, N)$,
$value(W_2, P)$.
$value(W_1, P)$ **if** $output(W_1, N)$,
$switches(N)$,
$input(W_2, N)$,
$value(W_2, P)$,
$destination(P, N_{dest})$,
$leads\_to(W_1, N_{dest})$.
end of *value*

action declarations

apply_signal : **action**
attributes
source_node : *source_nodes*
signal : *signals*
axioms
impossible *apply_signal* **if** $source\_node = Computer_1$,
$signal = Packet$,
$source(Packet, Computer_2)$,
$Computer_1 \neq Computer_2$.
end of *apply_signal*

And now an example of a system description for the Ethernet network module:

system description *ethernet_example*

declarations of *ethernet_example*

import *directed_multigraph* **from** *commonsense_library*

import *flow_network* **from** *commonsense_library*

import *ethernet_network* **from** *commonsense_library*

structure of *ethernet_example*

sorts

$hub \in hubs$
$switch \in switches$
$c_1, c_2, c_3, c_4, c_5, c_6 \in computers$
$w_1, w_2, w_3, w_4, w_5, w_6, w_7 \in wires$
$p_1, p_2, p_3 \in packets$

actions

instance $send\_packet(C, P)$ **where** $computers(C)$,
$packets(P)$: $apply\_signal$
$source\_node := C$
$signal := P$

instance $add\_link(W, N_1, N_2)$ **where** $wires(W)$: add_link
$arc := W$
$start := N_1$
$end := N_2$

instance $remove\_link(W, N_1, N_2)$ **where** $wires(W)$: $remove\_link$
$arc := W$
$start := N_1$
$end := N_2$

statics

$source(p_1, c_5)$.
$destination(p_1, c_3)$.
$source(p_2, c_2)$.
$destination(p_2, c_6)$.
$source(p_3, c_4)$.
$destination(p_3, c_1)$.
$\neg destination(p_1, C)$ **if** $C \neq c_3$.
$\neg destination(p_2, C)$ **if** $C \neq c_6$.
$\neg destination(p_3, C)$ **if** $C \neq c_1$.

The ASP encoding of this system description can be used to reason about the scenario represented by the following history:

$holds(links(c_1, w_1, hub), 0)$.
$holds(links(c_2, w_2, hub), 0)$.
$holds(links(c_3, w_3, hub), 0)$.
$holds(links(c_4, w_4, hub), 0)$.
$holds(links(c_5, w_5, switch), 0)$.
$holds(links(c_6, w_6, switch), 0)$.
$holds(links(switch, w_7, hub), 0)$.
$occurs(send\_packet(c_5, p_1), 0)$.
$occurs(remove\_link(w_7, switch, hub), 1)$.
$occurs(send\_packet(c_2, p_2), 2)$.

The packet p_1 will be received at the destination at time point 1. The packet p_2 however will never be received at its destination because the connection between its source and destination is interrupted before the packet is sent.

6.2 Blocks World Problem

In the section decribing the syntax of language $\mathcal{AL}_d$, we showed that executability conditions are statements of the type (3). So far, all our examples contained

executability conditions involving only one action ($k = 1$). We will illustrate the use of executability conditions involving more that one action ($k \geq 2$) by representing the Blocks World Problem.

Example 5. [Blocks World Problem]
Imagine a set of cubes(blocks) sitting on a table. The goal is to build one or more vertical stacks of blocks. The catch is that only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved. A block can only occupy a single location, and no block can directly support more than one other block.

The tops of blocks in this domain seem to be *atomic areas*: areas that cannot be further divided into smaller areas and can be occupied by at most one thing at a time. In the blocks world domain, a thing can not be located in two atomic areas simultaneously. Atomic areas can be used to represent other domains, such as the white and black squares on a chess board. Therefore, we can create a library module about atomic areas. Note that we invision this library module to be used together with the *basic_geography* module, which is part of the *motion* library module.

module *atomic_geography*

sort declarations

atomic_areas : *areas*
% an area that cannot be further divided into smaller areas,
% and can be occupied by at most one thing at a time

fluent declarations

$within(areas, areas)$: **static fluent**
axioms
$\neg within(A_1, A_2)$ **if** $atomic\_areas(A_2)$.
end of *within*

$disjoint(areas, areas)$: **static fluent**
axioms
$disjoint(A_1, A_2)$ **if** $atomic\_areas(A_1)$,
$atomic\_areas(A_2)$,
$A_1 \neq A_2$.
end of *disjoint*

$loc\_in(things, areas)$: **inertial fluent**
axioms
$\neg loc\_in(T_1, A)$ **if** $loc\_in(T_2, A)$,
$atomic\_areas(A)$,
$T_1 \neq T_2$.
end of *loc_in*

Now we can create the system description for the Blocks World Problem. We will make use of the *motion* and *atomic_geography* library modules. Note the use of indexes in the executability condition stating that only one instance of the *move_block* action can occur at each step. We will abstract away from the robotic arm, and consider blocks to be *movers*.

system description *blocks_world*

declarations of *blocks_world*

import $motion, atomic\_geography$ **from** *commonsense_library*

module *main*

sort declarations

$atomic\_areas$: $areas$
$blocks$: $movers$

fluent declarations

$is\_top(atomic\_areas, blocks)$: **static fluent**

axioms

$\neg is\_top(A_1, B)$ **if** $is\_top(A_2, B)$, $A_1 \neq A_2$.
$\neg is\_top(A, B_1)$ **if** $is\_top(A, B_2)$, $B_1 \neq B_2$.

end of *is_top*

action declarations

$move\_block$: $move$

axioms

impossible $move\_block$ **if** $actor = B_1$, $is\_top(A, B_1)$, $loc\_in(B_2, A)$.
impossible $move\_block$ **if** $actor = B$, $dest = A$, $is\_top(A, B)$.
impossible $move\_block_1, move\_block_2$.

end of *move_block*

structure of *blocks_world*

sorts

$b_1, b_2, b_3, b_4, b_5, b_6 \in blocks$
$table \in areas$
$top(b_1), top(b_2), top(b_3), top(b_4), top(b_5), top(b_6) \in atomic\_areas$

actions

instance $move\_block(B, A)$ **where** $blocks(B)$: $move\_block$
$actor := B$
$dest := A$

statics

$is\_top(top(B), B)$ **if** $blocks(B)$.
$disjoint(table, A)$ **if** $is\_top(A, B)$.
$\neg within(A, A)$.

6.3 Biology Example: Cell Cycle

The cell cycle refers to the phases a cell goes through from its "birth" to its division into two daughter cells. The cell consists of a number of parts e.g. nucleus, cytoplasm, etc. These immediate parts of the cell in turn consist of other parts, which form a "*part of*" hierarchy, say H. To model this hierarchy we introduce a sort called $classes\_of\_parts$ and the relations $father(C_1, C_2)$, where C_1 is the father class of class C_2 in H, and $root(C)$, where C is the root of H. We will be interested in the number of different parts present in the environment during different stages of the cell cycle. Therefore, the states of our domain will be described by an inertial fluent, $num(C_1, C_2, N)$, which holds if the number of parts from class C_1 in one part from the class C_2 is N (for instance, $num(nucleus, cell, 2)$ will indicate that at the current stage of the cell cycle, every cell in the environment has 2 nuclei). Now we need to describe the actions of our domain. For simplicity, we assume that the cell cycle consists of three consecutive steps: interphase, mitosis and cytokinesis. To describe this simplified cell cycle we will need two actions: *duplicate* and *split*. The action *duplicate*, which has an attribute *class* of sort $classes\_of\_parts$, doubles the number of every part from this class present in the environment. *Split*, which also has an attribute *class*, duplicates elements of this class and cuts in half the number of parts from the daugther classes of *class*. For instance, if the environment consists of one cell with two nuclei, *split*(*cell*) will increase the number of cells to two, each containing only one nucleus. In addition to these two actions we will have one exogenous action, *prevent*, which will nulify the effects of duplication and splitting for class C. The description of the domain will be given in a module *basic_cell_cycle* which can eventually become part of a more general *cell_cycle* library module.

module *basic_cell_cycle*

sort declarations

$classes\_of\_parts$: **sort**
$numbers$: **sort**
$even\_numbers$: $numbers$

fluent declarations

$father(classes\_of\_parts, classes\_of\_parts)$: **static fluent**

axioms

$\neg father(C_1, C)$ **if** $father(C_2, C)$, $C_1 \neq C_2$.
$\neg father(C_1, C)$ **if** $root(C)$.

end of $father$

$root(classes\_of\_parts)$: **static fluent**
axioms
$\neg root(C_1)$ **if** $root(C_2)$,
$C_1 \neq C_2$.
end of *root*

$num(classes\_of\_parts, classes\_of\_parts, numbers)$
: **inertial fluent**
axioms
$\neg num(C_1, C_2, N_2)$ **if** $num(C_1, C_2, N_1)$,
$N_1 \neq N_2$.
$num(C_3, C_1, N)$ **if** $father(C_1, C_2)$,
$num(C_2, C_1, N_1)$,
$num(C_3, C_2, N_2)$,
$C_3 \neq C_2$,
$N = N_1 * N_2$.
end of *num*

$prevented\_dupl(classes\_of\_parts)$: **inertial fluent**

action declarations

duplicate : **action**
attributes
$class : classes\_of\_parts$
axioms
duplicate **causes** $num(C_1, C_2, N_2)$ **if** $class = C_1$,
$father(C_2, C_1)$,
$num(C_1, C_2, N_1)$,
$N_2 = 2 * N_1$.
impossible *duplicate* **if** $class = C$,
$prevented\_dupl(C)$.
end of *duplicate*

split : *duplicate*
axioms
split **causes** $num(C_1, C_2, N_2)$ **if** $class = C_2$,
$father(C_2, C_1)$,
$num(C_1, C_2, N_1)$,
$even\_numbers(N_1)$,
$N_1 \neq 0$,
$N_1 = N_2 * 2$.
end of *split*

prevent_duplication : **action**
attributes
class : *classes_of_parts*
axioms
prevent_duplication **causes** *prevented_dupl*(C) **if** $class = C$.
end of *prevent_duplication*

As mentioned above, we assume that the cell cycle is a sequence of three consecutive phases: interphase, mitosis and cytokinesis. The following module represents knowledge about sequences of actions.

module *sequence*

sort declarations

elements : **sort**
sequences : **sort**
numbers : **sort**

fluent declarations

component(*elements*, *numbers*, *sequences*) : **static fluent**
axioms
$\neg component(E, N_1, S)$ **if** $component(E, N_2, S)$, $N_1 \neq N_2$.
$\neg component(E, N_1, S_1)$ **if** $component(E, N_2, S_2)$, $S_1 \neq S_2$.
end of *component*

length(*numbers*, *sequences*) : **static fluent**
axioms
$\neg length(N_1, S)$ **if** $length(N_2, S)$, $N_1 \neq N_2$.
end of *length*

Various system descriptions of $\mathcal{ALM}$ specifying this process on different levels of granularity will contain the *basic_cell_cycle* and *sequence* modules and will differ from each other only by their structure. First, we consider a model in which cell cycle is viewed as a sequence of three elementary actions: interphase, mitosis, and cytokinesis. We also limit our domain to cells contained in an experimental environment that we will call *sample*. This first refinement of the cell cycle will include the modules *basic_cell_cycle* and *sequence*, and:

structure of *cell_cycle*(1)

sorts

$sample, cell, nucleus \in classes\_of\_parts$
$cell\_cycle \in sequences$
$interphase, mitosis, cytokinesis \in elements$

actions

instance *interphase* : **action**
instance *mitosis* : *duplicate*
class := *nucleus*

instance *cytokinesis* : *split*
class := *cell*

statics

$father(sample, cell)$.
$father(cell, nucleus)$.
$root(sample)$.
$component(interphase, 1, cell\_cycle)$.
$component(mitosis, 2, cell\_cycle)$.
$component(cytokinesis, 3, cell\_cycle)$.
$length(3, cell\_cycle)$.

Suppose now that our initial sample consists of one cell with one nucleus. We would like to know the number of cells and nuclei in the sample after the end of cell cycle. The answer can be obtained from the answer set of a program consisting of the ASP translation of the *cell_cycle*(1) system description, a theory of intentions ([16], [17])[5], and the domain history. The history, $\mathcal{H}_1$, is written as:

$observed(num(cell, sample, 1), true, 0)$.
$observed(num(nucleus, cell, 1), true, 0)$.
$intend(cell\_cycle, 0)$.

The answer set will contain the last step 3 and the facts:

$holds(num(cell, sample, 2), 3)$
$holds(num(nucleus, sample, 2), 3)$
$holds(num(nucleus, cell, 1), 3)$

At the end of the cycle, the sample contains two cells with one nucleus each. Suppose now we learned that: (Q12.9) *"In some organisms mitosis occurs without cytokinesis occurring"* and we wanted to know how many nuclei will be contained by a cell from the sample at the end of cell cycle. To answer the question we simply expand the history by

$\neg happened(cytokinesis, I)$

for every step I. The corresponding answer set will now contain:

$holds(num(cell, sample, 1), 2)$
$holds(num(nucleus, sample, 2), 2)$
$holds(num(nucleus, cell, 2), 2)$

Let us now consider the following question: (Q12.15) *"A researcher treats cells with a chemical that prevents DNA synthesis. This treatment traps the cells in which part of the cell cycle?"* To answer this question the system will need to know more about the structure of the cell and that of the interphase and mitosis. The second refinement of the cell cycle provides this additional knowledge.

The following cell components will be added: *the chromosomes inside the nucleus, the chromatids that are part of the chromosomes, and the DNA inside the chromatids*. The interphase is a sequence $[g_1, s, g_2]$ where g_1 and g_2 are elementary actions and s is a sequence of two elementary actions: *DNA synthesis*, and *the creation of sister chromatids*. Mitosis is a sequence of five actions: *prophase, prometaphase, metaphase, anaphase, and telophase*. The treatment of the cells with the chemical is represented by an exogenous action that prevents the duplication of the DNA.

structure of $cell\_cycle(2)$

sorts

$sample, cell, nucleus, chromosome, chromatid, dna \in classes\_of\_parts$
$cell\_cycle, interphase, s, mitosis \in sequences$
$interphase, mitosis, cytokinesis, g1, s, g2, dna\_synthesis,$
$sister\_chromatids, prophase, prometaphase, metaphase, anaphase,$
$telophase \in elements$

actions

instance $g1$: **action**
instance $dna\_synthesis$: $duplicate$
$class := dna$

instance $sister\_chromatids$: $split$
$class := chromatid$

instance $g2$: **action**

instance $prophase$: **action**
instance $prometaphase$: **action**
instance $metaphase$: **action**

instance $anaphase$: $split$
$class := chromosome$

instance $telophase$: $split$
$class := nucleus$

instance $cytokinesis$: $split$
$class := cell$

instance $treatment$: $prevent\_duplication$
$class := dna$

statics

$father(sample, cell).$
$father(cell, nucleus).$
$father(nucleus, chromosome).$
$father(chromosome, chromatid).$
$father(chromatid, dna).$
$root(sample).$

$component(interphase, 1, cell\_cycle).$
$component(mitosis, 2, cell\_cycle).$
$component(cytokinesis, 3, cell\_cycle).$
$length(3, cell\_cycle).$

$component(g1, 1, interphase).$
$component(s, 2, interphase).$
$component(g2, 3, interphase).$
$length(3, interphase).$

$component(dna\_synthesis, 1, s).$
$component(sister\_chromatids, 2, s).$
$length(2, s).$

$component(prophase, 1, mitosis).$
$component(prometaphase, 2, mitosis).$
$component(metaphase, 3, mitosis).$
$component(anaphase, 4, mitosis).$
$component(telophase, 5, mitosis).$
$length(5, mitosis).$

We can now capture the scenario in question Q12.15 via the following history $\mathcal{H}_2$:

$observed(num(cell, sample, 1), true, 0).$
$observed(num(nucleus, cell, 1), true, 0).$
$observed(num(chromosome, nucleus, 46), true, 0).$
$observed(num(chromatid, chromosome, 1), true, 0).$
$observed(num(dna, chromatid, 1), true, 0).$
$intend(cell\_cycle, 0).$
$happened(treatment, 0).$

To answer our question we define a relation *trapped_in*:

$$
\begin{aligned}
trapped\_in(V1) \leftarrow\ & holds(component(V1, K, S), I),\\
& holds(component(V2, K1, S), I),\\
& ended(V1),\\
& \text{not } ended(V2),\\
& holds(elements(V1), I),\\
& holds(elements(V2), I),\\
& holds(sequences(S), I),\\
& K1 = K + 1.\\
ended(V) \leftarrow\ & holds(elements(V), I),\\
& ends(V, I).
\end{aligned}
$$

and add the rules above to the ASP encoding. The answer set of the resulting program will contain $trapped\_in(g1)$, where $g1$ is the answer to question Q12.15.

The second refinement of the cell cycle, $cell\_cycle(2)$, can also be used to answer the following question: (Q26) *"Neurons do not typically divide. What can you conclude about the cell cycle in neurons?*

a. These cells will be permanently arrested in G1 phase.
b. These cells will be permanently arrested in G2 phase.
c. These cells will undergo cytokinesis.
d. These cells will quickly enter S phase.
e. The duration of their cell cycle is long."

The following history, $\mathcal{H}_3$, captures this scenario. We consider all cells involved in this scenario to be neurons and represent the fact that neurons typically don't divide as a default: the duplication of all classes of parts is prevented, unless the neuron is abnormal.

$$
\begin{array}{l}
observed(num(cell, sample, 1), true, 0).\\
observed(num(nucleus, cell, 1), true, 0).\\
observed(num(chromosome, nucleus, 46), true, 0).\\
observed(num(chromatid, chromosome, 1), true, 0).\\
observed(num(dna, chromatid, 1), true, 0).\\
intend(cell\_cycle, 0).\\
holds(prevented\_dupl(C), 0) \leftarrow classes\_of\_parts(C),\\
\qquad\qquad\qquad\qquad\qquad\qquad \text{not } ab\_neuron.
\end{array}
$$

The possible answers (a)-(e) are represented using the definition of *trapped_in* from the previous example (Q12.15) and a rule defining the relation *permanently_arrested_in* as a synonym of *trapped_in*. As well, we say that a cell *entered a phase*, where the phase is represented as a sequence, if the first action of the sequence ended successfully:

$$
\begin{array}{l}
answer(a) \leftarrow permanently\_arrested\_in(g1).\\
answer(b) \leftarrow permanently\_arrested\_in(g2).\\
answer(c) \leftarrow ended(cytokinesis).\\
answer(d) \leftarrow enter(s).\\
answer(e) \leftarrow long\_cell\_cycle.\\
permanently\_arrested\_in(X) \leftarrow trapped\_in(X).\\
enter(S) \leftarrow first\_action(A, S),\\
\qquad\qquad\qquad action(A),\\
\qquad\qquad\qquad holds(sequences(S), I),\\
\qquad\qquad\qquad ended(A).\\
first\_action(A, S) \leftarrow action(A),\\
\qquad\qquad\qquad holds(sequences(S), I),\\
\qquad\qquad\qquad holds(component(A, 1, S), I).\\
first\_action(A, S) \leftarrow action(A),\\
\qquad\qquad\qquad holds(sequences(S), I),\\
\qquad\qquad\qquad holds(sequences(S1), I),\\
\qquad\qquad\qquad holds(elements(S1), I),\\
\qquad\qquad\qquad holds(component(S1, 1, S), I),\\
\qquad\qquad\qquad first\_action(A, S1).
\end{array}
$$

As expected, the answer set of the resulting program will contain $answer(a)$; it will not contain $answer(b)$, $answer(c)$, $answer(d)$ or $answer(e)$

We would like to consider now the following question: (Q45) *"The phases below describe several events that occur during the process of mitosis:*

(1) Attachment of double-stranded chromosomes to the spindle apparatus
(2) Formation of single-stranded chromosomes, which move to opposite ends of the cell
(3) Disintegration of the nuclear membrane
(4) Nuclear membrane formation around each set of chromosomes, forming two nuclei
(5) Synthesis of a spindle apparatus

Which of the following is true about a cell with a defect that prevents (4) from occurring?

a. It will be a haploid cell.
b. It will contain 4N chromosomes.
c. It will enter the G2 phase of the cell cycle.
d. It will stay in metaphase.
e. It will remain in prophase"

To answer this question the system will need to know *some* details about the different phases of the cell cycle, but *not all* the details included in the *cell_cycle*(2) representation. In the new *cell_cycle*(3) system description we can ignore some inner components of the cell (the chromatids and the DNA) and we can consider the second stage of the interphase, the s phase, to be an action, not a sequence. The N number in the question refers to the species-specific number of chromosomes contained in the nucleus of a reproductive cell for that species (23 for humans). A non-reproductive cell contains $2N$ chromosomes in its nucleus when it is formed (46 for humans). A *haploid cell* is a cell that contains N chromosomes in its nucleus.

structure of *cell_cycle*(3)

sorts

$sample, cell, nucleus, chromosome \in classes\_of\_parts$
$cell\_cycle, interphase, mitosis \in sequences$
$interphase, mitosis, cytokinesis, g1, s, g2, prophase, prometaphase,$
$metaphase, anaphase, telophase \in elements$

actions

instance $g1$: **action**
instance s : **action**
instance $g2$: **action**

instance $prophase$: **action**
instance $prometaphase$: **action**
instance $metaphase$: **action**

instance *anaphase* : *duplicate*
class := *chromosome*

instance *telophase* : *split*
class := *nucleus*

instance *cytokinesis* : *split*
class := *cell*

statics

$father(sample, cell)$.
$father(cell, nucleus)$.
$father(nucleus, chromosome)$.
$root(sample)$.

$component(interphase, 1, cell\_cycle)$.
$component(mitosis, 2, cell\_cycle)$.
$component(cytokinesis, 3, cell\_cycle)$.
$length(3, cell\_cycle)$.

$component(g1, 1, interphase)$.
$component(s, 2, interphase)$.
$component(g2, 3, interphase)$.
$length(3, interphase)$.

$component(prophase, 1, mitosis)$.
$component(prometaphase, 2, mitosis)$.
$component(metaphase, 3, mitosis)$.
$component(anaphase, 4, mitosis)$.
$component(telophase, 5, mitosis)$.
$length(5, mitosis)$.

The scenario in this question will be represented via the following history, $\mathcal{H}_4$, in which the defect in (4) is understood to mean that the duplication of the nucleus is prevented:

$species\_N\_num(23)$. % the N number for humans is 23
$holds(num(cell, sample, 1), 0)$.
$holds(num(nucleus, cell, 1), 0)$.
$holds(num(chromosome, nucleus, Num), 0) \leftarrow species\_N\_num(N), Num = 2 * N$.
$holds(prevented\_dupl(nucleus), 0)$.
$intend(cell\_cycle, 0)$.

The encoding of the possible answers to question (Q45) will define *entering a phase* as in the previous example, (Q26). *Stay in* and *remain in* a phase will be synonyms of the relation *trapped_in* for question (Q12.15). The following rules will also be added:

$answer(a) \leftarrow haploid$.
$answer(b) \leftarrow four\_N\_chromosomes$.

$answer(c) \leftarrow enter(g2).$
$answer(d) \leftarrow stay\_in(metaphase).$
$answer(e) \leftarrow remain\_in(prophase).$
$haploid \leftarrow holds(num(chromosome, cell, N), I),$
$\quad species\_N\_num(N),$
$\quad max\_step(I).$
$four\_N\_chromosomes \leftarrow holds(num(chromosome, cell, N1), I),$
$\quad species\_N\_num(N),$
$\quad N1 = 4 * N,$
$\quad max\_step(I).$

The answer set of the program will contain $answer(b)$ and no other answers, as expected. A modification of the program above can also be used to answer the following variant of question (Q45): (Q45f) *"Suppose a human cell has the above defect (i.e. a defect that prevents nuclear membrane formation around each set of chromosomes, forming two nuclei). How many chromosomes will it have at the start and end of mitosis?"* We will replace the encoding of possible answers for (Q45) with the following rules for (Q45f):

$starts(S, I) \leftarrow action(A),$
$\quad holds(sequences(S), I),$
$\quad holds(component(A, 1, S), I),$
$\quad occurs(A, I).$

$starts(S, I) \leftarrow holds(sequences(S), I),$
$\quad holds(sequences(S1), I),$
$\quad holds(elements(S1), I),$
$\quad holds(component(S1, 1, S), I),$
$\quad starts(S1, I).$

$num\_chromosomes\_before\_mitosis(N) \leftarrow starts(mitosis, I),$
$\quad holds(num(chromosome, cell, N), I).$
$num\_chromosomes\_after\_mitosis(N) \leftarrow ends(mitosis, I),$
$\quad holds(num(chromosome, cell, N), I).$

The answer set of this new program will contain $num\_chromosomes\_before\_mitosis(46)$ and $num\_chromosomes\_after\_mitosis(92)$ as expected.

Finally, a similar question: (Q45g) *"How many chromosomes does a rat cell have when it is formed?"* can be answered using the system description $cell\_cycle(3)$ and the history:

$species\_N\_num(21).$ % the N number for rats is 21
$holds(num(cell, sample, 1), 0).$
$holds(num(nucleus, cell, 1), 0).$
$holds(num(chromosome, nucleus, Num), 0) \leftarrow species\_N\_num(N),$
$\quad Num = 2 * N.$
$intend(cell\_cycle, 0).$

The question will be encoded using the rule:

$$final\_num(chromosome, cell, N) \leftarrow holds(num(chromosome, cell, N), I), \\ ends(cell\_cycle, I).$$

The answer set of this program will contain the correct answer $final\_num(chromosome, cell, 42)$.

7 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.

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)
16. Baral, C., Gelfond, M.: Reasoning about Intended Actions. In: Proceedings of AAAI-05, pp. 689–694. AAAI Press (2005)
17. Gelfond, M.: Going Places - Notes on a Modular Development of Knowledge about Travel. In: AAAI 2006 Spring Symposium Series, pp. 56–66 (2006)

Notes

[1] At first sight, this executability condition seems to be equivalent to:

$$\begin{array}{lll} \textbf{impossible } move & \textbf{if} & origin = A_1, \\ & & dest = A_2, \\ & & \neg disjoint(A_1, A_2). \end{array}$$

But this is not the case. Imagine that the structure of *basic_travel* also contains an object *france* of sort *areas*, such that $within(paris, france)$. If *michael* is located in *paris* at time step 0 and we have two instances:

$$\begin{array}{l} \textbf{instance } move(michael, paris, france) \ : \ move \\ \quad actor := michael \\ \quad origin := paris \\ \quad dest := france \end{array}$$

$$\begin{array}{l} \textbf{instance } move(michael, france) \ : \ move \\ \quad actor := michael \\ \quad dest := france \end{array}$$

then the axiom that is currently in *basic_travel* will prevent both action instances *move(michael, paris, france)* and *move(michael, france)* from being executed, whereas the axiom in this endnote will only prevent the execution of *move(michael, paris, france)* and will allow the execution of *move(michael, france)*. The axiom currently in *basic_travel* expresses better the intended executability condition on action *move* because it doesn't rely on an attribute that can be omitted, such as *origin*.

[2] By making $m(aclasses(L))$ a bijection we ensure that each action class in $aclasses(L)$ is mapped into an action instance, and more importantly, that action class names with different indexes are mapped into different action instances.

[3] This axiom is introduced here only to illustrate the semantics of executability conditions of $\mathcal{ALM}$. It is not needed in the actual system description *basic_travel*, as other state constraints and executability conditions of *basic_travel* prevent such a scenario from occurring

[4] In order to illustrate the importance of this axiom, let's consider a scenario in which the monkey is located in place l_2 in the initial situation, where l_2 is not under the bananas and it is not the top of the box either. Without this axiom, an ASP planning module added to the ASP translation of this system description could find a plan in which: the monkey grips the box at step 0 causing the monkey to be holding the box at step 1; in step 1, as it is holding the box, the monkey moves to l_4, the top of the box (the monkey jumps on the box); in step 2, while located on the box and holding the box, it carries the box to l_1, under the bananas; in step 3, the monkey grips

the bananas. This is not an intended plan for the Monkey and Banana Problem. The new axiom added to the action class *move* prevents such trajectories from becoming part of the system description.

[5]Our theory of intentions will contain the following rules:

$$
\begin{array}{ll}
occurs(A, I) & \leftarrow intend(A, I), \\
 & \quad \text{not } \neg occurs(A, I). \\
intend(A, I1) & \leftarrow intend(A, I), \\
 & \quad \neg occurs(A, I), \\
 & \quad \text{not } \neg intend(A, I1), \\
 & \quad I1 = I + 1. \\
intend(V, I) & \leftarrow intend(S, I), \\
 & \quad holds(component(V, 1, S), I), \\
 & \quad holds(elements(V), I), \\
 & \quad holds(sequences(S), I). \\
intend(V2, I2) & \leftarrow intend(S, I1), \\
 & \quad holds(component(V2, K1, S), I), \\
 & \quad holds(component(V1, K, S), I), \\
 & \quad K1 = K + 1, \\
 & \quad ends(V1, I2), \\
 & \quad holds(elements(V1), I), \\
 & \quad holds(elements(V2), I) \\
 & \quad holds(sequences(S), I). \\
ends(S, I) & \leftarrow holds(sequences(S), I), \\
 & \quad holds(elements(V), I), \\
 & \quad holds(length(N, S), I), \\
 & \quad holds(component(V, N, S), I), \\
 & \quad ends(V, I). \\
ends(A, I1) & \leftarrow occurs(A, I), \\
 & \quad I1 = I + 1.
\end{array}
$$