Modular Action Language \textit{ALM}

DANIELA INCLEZAN  
\textit{Department of Computer Science and Software Engineering, Miami University}  
Oxford, OH 45056, USA  
(e-mail: inclezd@MiamiOH.edu)

MICHAEL GELFOND  
\textit{Department of Computer Science, Texas Tech University}  
Lubbock, TX 79409, USA  
(e-mail: michael.gelfond@ttu.edu)

Abstract

The paper introduces a new modular action language, \textit{ALM}, and illustrates the methodology of its use. It is based on the approach of Gelfond and Lifschitz (1993; 1998) in which a high-level action language is used as a front end for a logic programming system description. The resulting logic programming representation is used to perform various computational tasks. The methodology based on existing action languages works well for small and even medium size systems, but is not meant to deal with larger systems that require \textit{structuring of knowledge}. \textit{ALM} is meant to remedy this problem. Structuring of knowledge in \textit{ALM} is supported by the concepts of \textit{module} (a formal description of a specific piece of knowledge packaged as a unit), \textit{module hierarchy}, and \textit{library}, and by the division of a system description of \textit{ALM} into two parts: \textit{theory} and \textit{structure}. A \textit{theory} consists of one or more modules with a common theme, possibly organized into a module hierarchy based on a \textit{dependency relation}. It contains declarations of sorts, attributes, and properties of the domain together with axioms describing them. Structures are used to describe the domain’s objects. These features, together with the means for defining classes of a domain as special cases of previously defined ones, facilitate the stepwise development, testing, and readability of a knowledge base, as well as the creation of knowledge representation libraries. To appear in Theory and Practice of Logic Programming (TPLP).

\textit{KEYWORDS:} logic programming, reasoning about actions and change, action language

1 Introduction

In this paper we introduce a new modular action language, \textit{ALM}, and illustrate the principles of its use. Our work builds upon the methodology for representing knowledge about discrete dynamic systems introduced by Gelfond and Lifschitz (1993 1998). In this approach, a system is viewed as a \textit{transition diagram} whose nodes correspond to possible states of the system and whose arcs are labeled by actions. The diagram is defined by a \textit{system description} – a collection of statements in a high-level \textit{action language} expressing the direct and indirect effects of actions as well as their executability conditions (see, for instance, action languages \textit{A} (Gelfond and Lifschitz 1993), \textit{B} (Gelfond and Lifschitz 1998); \textit{AL} (Turner 1997; Baral and
Daniela Inclezan and Michael Gelfond

The non-modular extension of $\mathcal{AL}$ with multi-valued fluents (Dovier et al. 2007); $\mathcal{C}$ (Giunchiglia and Lifschitz 1998); $\mathcal{C}+$ (Giunchiglia et al. 2004a); $\mathcal{K}$ (Eiter et al. 2004); $\mathcal{D}$ (Strass and Thielscher 2012); $\mathcal{E}$ (Kakas and Miller 1997); $\mathcal{H}$ (Chintabathina et al. 2003; Chintabathina 2012). Such languages allow concise representations of very large diagrams. In order to reason about the system, its action language description is often translated into a logic program under the answer set semantics (Gelfond and Lifschitz 1988; 1991). This allows for the use of Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Niemelä 1998; Marek and Truszczynski 1999) to perform complex reasoning tasks such as planning, diagnosis, etc. This methodology was successfully used in a number of interesting medium size applications, but does not seem to be fully adequate for applications requiring a larger body of knowledge about actions and their effects, step-wise design, and multiple use of, possibly previously designed, pieces of knowledge. (The phenomenon is of course well known in Computer Science. Similar considerations led to the early development of notions of subroutine and module in procedural programming. In logic programming, early solutions were based on the concepts of macro and template (Baral et al. 2006; Calimeri and Ianni 2006).) Just a few examples of domains that we consider large enough to benefit from the above-mentioned practices are: the Zoo World and Traffic World examples proposed by Erik Sandewall (Sandewall 1999) and modeled in (Henschel and Thielscher 1999; Akman et al. 2004); the Monkey and Banana Problem by John McCarthy (McCarthy 1963; McCarthy 1968) and formalized in (Erdogan and Lifschitz 2006; Erdogan 2008); the Missionaries and Cannibals Problem by John McCarthy (McCarthy 1998) represented in (Gustafsson and Kvarnström 2004; Erdogan 2008).

This inadequacy is due to the fact that most action languages, with some notable exceptions like MAD (Lifschitz and Ren 2006; Erdogan and Lifschitz 2006; Desai and Singh 2007) and TAL-C (Gustafsson and Kvarnström 2004), have no built-in features for supporting the description of a domain’s ontology and its objects, and for structuring knowledge and creating knowledge-based libraries. $\mathcal{AL}_M$ is designed to address these problems. It is based on an earlier action language, $\mathcal{AL}$, introduced in (Gelfond and Inclezan 2009) where it is called $\mathcal{AL}_d$, which so far has been the authors’ language of choice (see, for instance, (Gelfond and Kahl 2014)). However, the basic ideas presented in the paper can be used for defining versions of $\mathcal{AL}_M$ based on other action languages.

$\mathcal{AL}_M$ has constructs for representing sorts (i.e., classes, kinds, types, categories) of objects relevant to a given domain, their attributes\(^1\) and a subsort relation that can be viewed as a directed acyclic graph (DAG). We refer to this relation as a sort hierarchy. These constructs support a methodology of knowledge representation that starts with determining the sorts of objects in a domain and formulating the domain’s causal laws and other axioms in terms of these sorts. The specialization construct of the language, which corresponds to the links of the sort hierarchy,

\(^1\) Attributes are intrinsic properties of a sort of objects. In $\mathcal{AL}_M$ they are represented by possibly partial functions defined on elements of that sort.
Modular Action Language $\mathcal{ALM}$ allows to define new sorts (including various sorts of actions) in terms of other, previously defined sorts.

The definition of particular objects populating the sorts is usually given only when the domain knowledge is used to solve a particular task, e.g., predicting the effects of some particular sequences of actions, planning, diagnosis, etc.

It is worth noting that allowing definitions of actions as special cases of other, previously defined actions was one of the main goals of actions languages like $\mathcal{ALM}$ and $\mathcal{MAD}$. Such definitions are not allowed in traditional action languages. $\mathcal{ALM}$'s solution consists in allowing action sorts, which do not exist in $\mathcal{MAD}$. We believe that the $\mathcal{ALM}$ solution is simpler than the one in $\mathcal{MAD}$, where special cases of actions are described using import statements (similar to bridge rules in C++).

$\mathcal{ALM}$ also facilitates the introduction of particular domain objects (including particular actions) that are defined as instances of the corresponding sorts. For example, an action $go(bob, london, paris)$ can be defined as an instance of action sort $move$ with attributes $actor$, $origin$, and $destination$ set to $bob$, $london$, and $paris$ respectively; action $go(bob, paris)$ is another instance of the same sort in which the origin of the $move$ is absent. Note that, since axioms of the domain are formulated in terms of sorts and their attributes, they are applicable to both of these actions. This is very different from the traditional action language representation of objects as terms, which requires separate axioms for $go(bob, london, paris)$ and $go(bob, paris)$.

Structuring of knowledge in $\mathcal{ALM}$ is supported by the concepts of module, module hierarchy, and library, and by the division of a system description of $\mathcal{ALM}$ into two parts: theory and structure. Theories contain declarations of sorts, attributes, and properties of the domain together with axioms describing them, while structures are used to describe the domain’s objects. Rather traditionally, $\mathcal{ALM}$ views a module as a formal description of a specific piece of knowledge packaged as a unit. A theory consists of one or more modules with a common theme, possibly organized into a module hierarchy based on a dependency relation. Modules of a theory can be developed and tested independently, which facilitates the reuse of knowledge and stepwise development and refinement (Wirth 1971) of knowledge bases, and increases their elaboration tolerance (McCarthy 1998).

Theories describing recurrent knowledge may be stored in libraries and used in different applications. The structure part of an $\mathcal{ALM}$ system description contains definitions of objects of the domain together with their sorts, values of their attributes, and statics - relations between objects that cannot be changed by actions. If a system description of $\mathcal{ALM}$ satisfies some natural consistency requirements and provides complete information about the membership of its objects in the system’s sorts then it describes the unique transition diagram containing all possible trajectories of the system. In this sense $\mathcal{ALM}$ is semantically similar to $\mathcal{AL}$. There are also some substantial differences. First, if no complete information about membership of objects in sorts is given, then the system description specifies the collection of transition diagrams corresponding to various possible placements of objects in the system’s sorts. This has no analog in $\mathcal{AL}$. Second, in addition to the semantics of its system descriptions, $\mathcal{ALM}$ provides semantics for its theories. Informally,
a theory of $\mathcal{ALM}$ can be viewed as a function taking as an input objects of the domain, their sort membership, and the values of static relations, and returning the corresponding transition diagram – a possible model of the theory. (This definition has some similarity with the notions of module developed for logic programs under the answer set semantics, e.g. (Oikarinen and Janhunen 2006) and (Lierler and Truszczynski 2013). Accurate mathematical analysis of these similarities and their use for automatic reasoning in $\mathcal{ALM}$ is a matter for future research.) The availability of a formal semantics clarifies the notion of an $\mathcal{ALM}$ theory and allows us to define an entailment relation ($T$ entails $q$ if $q$ is true in every model of $T$).

To accurately define the semantics of $\mathcal{ALM}$ theories, we introduce the notion of a basic action theory ($\mathcal{BAT}$) — a pair consisting of a specific type of sorted signature (which we call an action signature), and a set of axioms over this signature. An interpretation $I$ of the signature of a $\mathcal{BAT}$ theory $T$ defines: objects, their sort membership, and statics; while $T$ can be viewed as a function that takes $I$ as input and returns the transition diagram $T(I)$ defined by $I$. In a sense, $T(I)$ is very similar to system descriptions of $\mathcal{AL}$ and other traditional action languages. The difference is in the forms of their signatures and axioms. As in $\mathcal{AL}$, the precise definition of states and transitions of $T(I)$ is given in terms of its translation into logic programs under the answer set semantics.

A system description $D$ of $\mathcal{ALM}$ can be viewed as a formal definition of a particular $\mathcal{BAT}$ theory $T$, and a class of its interpretations. The latter is given by the structure of $D$, the former by its theory. If the structure of $D$ is complete, i.e., defines exactly one interpretation $I$, then $D$ represents $T(I)$.

An earlier version of $\mathcal{ALM}$ has been tested in the context of a real-life application, as part of our collaboration on Project Halo. Project Halo is a research effort by Vulcan Inc. aimed towards the development of a Digital Aristotle – “an application containing large volumes of scientific knowledge and capable of applying sophisticated problem-solving methods to answer novel questions” (Gunning et al. 2010). The Digital Aristotle uses the knowledge representation language called SILK (Semantic Inferencing on Large Knowledge) (Grosof et al. 2009), which is based on the well-founded semantics (Van Gelder et al. 1991) and transaction logic with defaults and argumentation theories (Fodor and Kifer 2011). Our first contribution to Project Halo consisted in creating an $\mathcal{ALM}$ formalization of an important biological process, cell division (Inclezan and Gelfond 2011). The use of $\mathcal{ALM}$ allowed us to create libraries of knowledge and reuse information when representing the cell division domain. As a second step, we created a question answering system capable of answering complex temporal projection questions about this biological process (Inclezan 2010). Our model of cell division represented in the higher level language $\mathcal{ALM}$ served as a front end for the question answering system, which was implemented both in ASP and in the language of the Digital Aristotle.

Our language has evolved since our collaboration on Project Halo. The version of $\mathcal{ALM}$ presented here differs from that described in previous papers (Gelfond and Inclezan 2009) and (Inclezan and Gelfond 2011) in various ways. We simplified and generalized the basic concepts of our language, as well as its syntax and semantics. (We say more about the new features of $\mathcal{ALM}$ in the conclusion section of the paper.)
The reasoning in ALM is based on the reduction of temporal projection, planning, diagnosis, etc. to the problem of computing the answer sets of logic programs (for a general description see, for instance, Baral 2003) by ASP solvers (see Niemelä and Simons 1997, Gebser et al. 2012, or Leone et al. 2006).

The rest of this paper is organized as follows: we first introduce the concept of basic action theory, which is a fundamental concept in this work. We then describe language ALM and the methodology of ALM’s use. We end with conclusions and future work. There are three appendices containing the grammar of ALM, the description of the use of ALM in Digital Aristotle, and a comparison between ALM and MAD.

2 Basic Action Theories

In this section we give the definition of a fundamental concept of ALM called basic action theory (BAT). A BAT consists of a collection of axioms over a so called action signature — a special type of sorted signature providing suitable vocabulary for representing knowledge about dynamic domains. Sorted signatures needed for our purpose are somewhat atypical. They allow partial functions and contain means for describing a hierarchy of sorts and attributes of their elements. We start with the precise definition of sorted signatures and their interpretations.

2.1 Sorted Signatures and Their Interpretations

By sorted signature we mean a tuple

$\Sigma = \langle C, O, H, F \rangle$

where $C$, $O$, and $F$ are sets of strings over some fixed alphabet. The strings are used to name sorts, objects, and (possibly partial) functions respectively. Each function symbol $f \in F$ is assigned a positive integer $n$ (called $f$’s arity), sorts $c_0, \ldots, c_n$ for its parameters, and sort $c$ for its values. We refer to $c$ as the range of $f$ and use the standard mathematical notation $f: c_0 \times \ldots \times c_n \rightarrow c$ for this assignment.

Finally, $H$ is a sort hierarchy — a directed acyclic graph with two types of nodes: sort nodes labeled by sort names from $C$, and object nodes labeled by object names from $O$. Whenever convenient we identify nodes of the hierarchy with their labels. A link from sort $c_1$ to sort $c_2$, denoted by $\langle c_1, c_2 \rangle$, indicates that elements of sort $c_1$ are also elements of sort $c_2$. We refer to $c_2$ as a parent of $c_1$. A link from object $o$ to a sort $c$, denoted by $\langle o, c \rangle$, indicates that object $o$ is of sort $c$. For simplicity, we assume that the graph has exactly one sink node, which corresponds to the sort containing all the elements of the hierarchy. A triple $\langle C, O, H \rangle$ will be sometimes referred to as an ontology.

Sorts, object constants, and functions of a sorted signature are normally partitioned into user-defined, pre-defined, and special.

The collection of pre-defined symbols may include names for some commonly used sorts and functions, such as: sorts booleans and integers; a sort $[m..n]$ for every pair of natural numbers $m$ and $n$ such that $m < n$, denoting the set of
natural numbers in the closed interval $[m, n]$; standard object constants $\text{true}$, $\text{false}$, 0, 1, 2, etc., denoting elements of these sorts; standard arithmetic functions and relations $+$, $-$, $\ast$, $/$, $\text{mod}$, $<$, $\leq$, etc. (The list is not exhaustive. When needed we may introduce other similar symbols.) All these symbols are pre-interpreted, i.e., come with their usual mathematical interpretations.

The collection of special symbols consists of:

- sorts and function symbols pertinent to sort hierarchies of sorted signatures:
  - Sort $\text{nodes}$ denoting the collection of sorts labeling the sort nodes of $\mathcal{H}$. This sort is never used as a label of a node in $\mathcal{H}$.
  - Sort $\text{object\_constants}$ denoting the collection of constants labeling the object nodes of $\mathcal{H}$. This sort is never used as a label of a node in $\mathcal{H}$.
  - Sort $\text{universe}$ denoting the collection of elements of sorts from $\mathcal{H}$;
  - Function symbol $\text{link} : \text{nodes} \times \text{nodes} \to \text{booleans}$ where $\text{link}(c_1, c_2)$ returns $\text{true}$ iff $\mathcal{H}$ contains a link from sort $c_1$ to sort $c_2$.
  - Function symbol $\text{is\_a} : \text{universe} \times \text{nodes} \to \text{booleans}$ where $\text{is\_a}(x, c)$ returns $\text{true}$ if $c$ is a source node of $\mathcal{H}$ (i.e., $c$ has no subsorts in $\mathcal{H}$) and object $x$ from the universe is of the sort denoted by $c$.
  - Function symbol $\text{instance} : \text{universe} \times \text{nodes} \to \text{booleans}$ denoting the membership relation between objects of the universe and the sorts of the domain. This function will be later defined in terms of function $\text{is\_a}$.
  - Function symbols $\text{subsort} : \text{nodes} \times \text{nodes} \to \text{booleans}$, $\text{has\_child}$, $\text{has\_parent}$, $\text{sink}$, $\text{source} : \text{nodes} \to \text{booleans}$ describing properties of sorts of $\mathcal{H}$ and their members. All these functions (with their self-explanatory meaning) will be later defined in terms of function $\text{link}$.
- Function symbol $\text{dom}_f : c_0 \times \ldots \times c_n \to \text{booleans}$ (read as domain of $f$) for every user-defined function symbol $f : c_0 \times \ldots \times c_n \to c$ with $n > 0$.

Terms of a sorted signature are defined as usual:

- A variable and an object constant is a term.
- If $f : c_0 \times \ldots \times c_n \to c$ is a function symbol and $t_0, \ldots, t_n$ are terms then $f(t_0, \ldots, t_n)$ is a term.

Expressions of the form

$$t_1 = t_2 \quad \text{and} \quad t_1 \neq t_2$$

are called literals. Positive literals are also referred to as atoms. (For simplicity of presentation we use standard shorthands and write $t$ and $\neg t$ instead of $t = \text{true}$ and $t = \text{false}$, respectively; $3 \leq 5$ instead of $\leq (3, 5)$; etc.) Terms and literals not containing variables are called ground. Our notion of an interpretation of a sorted signature is slightly different from the traditional one.
A non-empty set \(|\mathcal{I}|\) of strings called the universe of \(\mathcal{I}\).

An assignment that maps

- every user-defined sort \(c\) of \(\mathcal{H}\) into a subset \(\mathcal{I}(c)\) of \(|\mathcal{I}|\) and user defined object constant \(o\) into an element from \(|\mathcal{I}|\);
- every user-defined function symbol \(f : c_0 \times \ldots \times c_n \to c\) of \(\Sigma\) into a (possibly partial) function \(\mathcal{I}(f) : \mathcal{I}(c_0) \times \ldots \times \mathcal{I}(c_n) \to \mathcal{I}(c)\);
- the special function \(\text{is}_a\) into function \(\mathcal{I}(\text{is}_a)\) such that:
  - for every \(x \in |\mathcal{I}|\) and every sort \(c\) of \(\mathcal{H}\), \(\mathcal{I}(\text{is}_a)(x, c)\) is true iff \(c\) is a source node of \(\mathcal{H}\) and \(x \in \mathcal{I}(c)\) and
  - for every object \(o\) and sort \(c\) of \(\mathcal{H}\), \(\mathcal{I}(\text{is}_a)(\mathcal{I}(o), c)\) is true iff \((o, c) \in \mathcal{H}\);
- the special function \(\text{link}\) into function \(\mathcal{I}(\text{link})\) such that for every two sort nodes \(c_1, c_2\), \(\mathcal{I}(\text{link})(c_1, c_2)\) is true iff \(|c_1, c_2| \in \mathcal{H}|\);
- the special function \(\text{dom}_f\) for user-defined function \(f : c_0 \times \ldots \times c_n \to c\) into function \(\mathcal{I}(\text{dom}_f)\) such that for every \(\bar{x} \in \mathcal{I}(c_0) \times \ldots \times \mathcal{I}(c_n)\), \(\mathcal{I}(\text{dom}_f)(\bar{x})\) is true iff \(\bar{x}\) belongs to the domain of \(\mathcal{I}(f)\).

- On pre-defined symbols, \(\mathcal{I}\) is identified with the symbols’ standard interpretations.

An interpretation \(\mathcal{I}\) of \(\Sigma\) can be naturally extended to ground terms: if \(\mathcal{I}\) is defined on terms \(t_1, \ldots, t_n\) and \(\mathcal{I}(f)\) is defined on the tuple \((\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))\) then

\[
\mathcal{I}(f(t_1, \ldots, t_n)) = \text{def } \mathcal{I}(f)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n));
\]

Otherwise \(\mathcal{I}(f(t_1, \ldots, t_n))\) is undefined.

Finally, we say that an atom \(t_1 = t_2\) is

- true in \(\mathcal{I}\) if both \(\mathcal{I}(t_1)\) and \(\mathcal{I}(t_2)\) are defined and have the same value;
- false in \(\mathcal{I}\) if both \(\mathcal{I}(t_1)\) and \(\mathcal{I}(t_2)\) are defined and have different values; and
- undefined in \(\mathcal{I}\) otherwise.

Similarly, a literal \(t_1 \neq t_2\) is true in \(\mathcal{I}\) if \(t_1 = t_2\) is false in \(\mathcal{I}\); it is false in \(\mathcal{I}\) if \(t_1 = t_2\) is true in \(\mathcal{I}\); and undefined otherwise.

Note that every interpretation \(\mathcal{I}\) can be uniquely represented by the collection of atoms that are true in this interpretation. For instance, for every sort \(c\) of \(\mathcal{H}\), \(\mathcal{I}(c)\) can be represented as the set \(\{\text{instance}(o, c) : \mathcal{I}(o) \in \mathcal{I}(c)\}\); for a unary function \(f\), \(\mathcal{I}(f)\) can be viewed as the set \(\{f(x) = y : \mathcal{I}(x) \in \mathcal{I}(\text{dom}_f)\\text{ and } \mathcal{I}(f)(\mathcal{I}(x)) = \mathcal{I}(y)\}\), etc.

### 2.2 Action Signature and Axioms of a SAT

Since \(\mathcal{ALM}\) is a language for specifying properties of actions, in what follows we limit ourselves to action signatures — sorted signatures that
• contain a special sort actions and
• have their user-defined and special function symbols divided into three disjoint categories: attributes, statics, and fluents. Attributes describe intrinsic properties of objects of a given sort; statics and fluents describe relations between objects. Values of attributes and statics are constants – they cannot be changed by actions. The values of fluents can. Both statics and fluents are further divided into basic and defined. The latter are total boolean functions that can be defined in terms of the former. They are used primarily for the brevity of representation.

A literal (atom) in which \( f \) is an attribute is called an attribute literal (atom). Similarly for static and fluent literals that are, in turn, divided into basic and defined. We assume that all special functions of an action signature, except dom\( f \), are defined statics; dom\( f \) is a basic fluent when \( f \) is a basic fluent and a defined static otherwise. Since the semantics of \( \mathcal{ALM} \) will be defined in terms of a version of ASP with function symbols, ASP\{\( f \}\} [Balduccini 2013], which does not allow terms with nested user-defined functions, we limit atoms of an action signature to those constructed from terms with at most one user-defined function symbol. (This is not a serious limitation and can easily be avoided by viewing nested terms as shorthands.)

We can now define the syntax and informal semantics of statements of a \( \mathcal{BAT} \) over a fixed action signature \( \Sigma \). Variables in these statements are universally quantified.

Definition 2 (Statements of a \( \mathcal{BAT} \))

- A dynamic causal law is an expression of the form
  \[
  \text{occurs}(a) \text{ causes } f(\bar{x}) = o \text{ if } \text{instance}(a,c), \text{cond} \tag{2}
  \]
  where \( a \) and \( o \) are variables or object constants, \( f \) is a basic fluent, \( c \) is the sort actions or a sub-sort of it, and \( \text{cond} \) is a collection of literals. The law says that an occurrence of an action \( a \) of the sort \( c \) in a state satisfying property \( \text{cond} \) causes the value of \( f(\bar{x}) \) to become \( o \) in any resulting state.
- A state constraint is an expression of the form
  \[
  f(\bar{x}) = o \text{ if } \text{cond} \tag{3}
  \]
  where \( o \) is a variable or an object constant, \( f \) is any function except a defined function, and \( \text{cond} \) is a collection of literals. The law says that the value of \( f(\bar{x}) \) in any state satisfying condition \( \text{cond} \) must be \( o \). Additionally, \( f(\bar{x}) = o \) can also be replaced by the object constant \text{false}, in which case the law says that there is no state satisfying condition \( \text{cond} \).
- The definition of a defined function \( p \) is an expression of the form
  \[
  p(\bar{t}_1) \text{ if } \text{cond}_1 \]
  \[
  \ldots
  p(\bar{t}_k) \text{ if } \text{cond}_k \tag{4}
  \]
  where \( \bar{t}s \) are sequences of terms, and \( \text{cond}_1, \ldots, \text{cond}_k \) are collections of literals. Moreover, if \( p \) is a static then \( \text{cond}_1, \ldots, \text{cond}_k \) can not contain fluent
literals. Statements of the definition will be often referred to as its clauses. The statement says that, for every \( Y \), \( p(Y) \) is true in a state \( \sigma \) iff there is \( 1 \leq m \leq k \) such that statements \( \text{cond}_m \) and \( t_m = Y \) are true in \( \sigma \).

- An executability condition for actions is an expression of the form
  \[
  \text{impossible} \ \text{occurs}(a) \ \text{if} \ \text{instance}(a, c), \ \text{cond}
  \]
  (5)
  where \( a \) is a variable or an object constant, \( c \) is the sort \( \text{actions} \) or a subsort of it, and \( \text{cond} \) is a collection of literals and expressions of the form \( \text{occurs}(t) \) or \( \neg \text{occurs}(t) \) where \( t \) is a variable or an object constant of the sort \( \text{actions} \).

The law says that an occurrence of an action \( a \) of the sort \( c \) is impossible when condition \( \text{cond} \) holds.

Dynamic causal laws and constraints will be sometimes referred to as causal laws. We use the term head to refer to \( l \) in (2) and (3), and to any of the \( p(t_i) \), \( 1 \leq i \leq k \), in (4). We call body the expression to the right of the keyword if in statements (2), (3), (5), or in any of the statements of (4). Statements not containing variables will be referred to as ground.

**Definition 3 (Basic Action Theory – BAT)**

A Basic Action Theory (BAT) is a pair consisting of an action signature \( \Sigma \) and a collection \( T \) of statements over \( \Sigma \) (called axioms of the theory) such that:

- If \( f \) is a basic fluent then
  - \( T \) contains a state constraint:
    \[
    \text{dom}_f(X_0, \ldots, X_n) \ \text{if} \ f(X_0, \ldots, X_n) = Y
    \]
    (6)
    — No dynamic causal law of \( T \) contains an atom formed by \( \text{dom}_f \) in the head.
  
- If \( f \) is a defined fluent, a static, or an attribute then \( T \) contains the definition:
  \[
  \text{dom}_f(X_0, \ldots, X_n) \ \text{if} \ f(X_0, \ldots, X_n) = Y
  \]
  (7)

- \( T \) contains definitions of special statics of the hierarchy given in terms of functions \( \text{is} \_ \_ \_ a \) and \( \text{link} \):
  \[
  \begin{align*}
  \text{instance}(O, C) & \ \text{if} \ \text{is} \_ \_ \_ a(O, C) \\
  \text{instance}(O, C_2) & \ \text{if} \ \text{instance}(O, C_1), \ \text{link}(C_1, C_2) \\
  \text{has} \_ \text{child}(C_2) & \ \text{if} \ \text{link}(C_1, C_2) \\
  \text{has} \_ \text{parent}(C_1) & \ \text{if} \ \text{link}(C_1, C_2) \\
  \text{source}(C) & \ \text{if} \ \neg \text{has} \_ \text{child}(C) \\
  \text{sink}(C) & \ \text{if} \ \neg \text{has} \_ \text{parent}(C) \\
  \text{subsort}(C_1, C_2) & \ \text{if} \ \text{link}(C_1, C_2) \\
  \text{subsort}(C_1, C_2) & \ \text{if} \ \text{link}(C_1, C), \ \text{subsort}(C, C_2)
  \end{align*}
  \]
  (8)

To simplify the notation, in what follows we will often identify a theory with the collection of its axioms. Axioms (6)–(8) above are self-explanatory, with the possible exception of the restriction prohibiting the appearance of \( \text{dom}_f \) in the head of dynamic causal laws. To understand the latter requirement it is sufficient to notice
that it is not enough to include object $O$ in the domain of basic fluent $f$ — it is also necessary to specify the value of $f(O)$. Otherwise the causal law making $\text{dom}_f(O)$ true would become non-deterministic, which is not allowed in the current version of $\mathcal{ALM}$. The presence of a law assigning a value to $f(O)$ makes dynamic causal laws with $\text{dom}_f$ in the head unnecessary. It is however useful to allow dynamic causal laws with $\neg \text{dom}_f(O)$ in the head as a simple way of removing $O$ from the domain of $f$.

The following is an example of a basic action theory.

Example 1 (A Basic Action Theory $T^0$)
Let us consider an action signature $\Sigma^0$ with three sorts, $c_1$, $c_2$ and $c_3$, the special sorts universe and actions, and the pre-defined sort booleans, organized in a hierarchy $\mathcal{H}^0$ in which universe is the parent of $c_1$, $c_1$ is the parent of $c_2$, $c_3$, actions, and booleans, and object constant $o$ is of sort $c_3$; attributes $\text{attr}_1, \text{attr}_2 : \text{actions} \rightarrow c_3$;

![Fig. 1. Hierarchy $\mathcal{H}^0$ of $T^0$](image)

basic fluents $f, g : c_2 \rightarrow c_3$; and special functions like $\text{link}, \text{is}_a, \text{dom}_f, \text{dom}_g$. The hierarchy $\mathcal{H}^0$ can be seen in Figure 1 but we omitted from the picture the sort universe whose only child is $c_1$.

The basic action theory $T^0$ over $\Sigma^0$ consists of the causal laws

$\text{occurs}(A) \text{ causes} f(X) = Y$ if $\text{instance}(A, \text{actions}),
\text{attr}_1(A) = Y,$
$g(X) = o$

$\text{occurs}(A) \text{ causes} \neg \text{dom}_f(X)$ if $\text{instance}(A, \text{actions}),
\text{attr}_2(A) = o$

false if $\neg \text{dom}_g(X),
\text{instance}(X, c_2)$.

The third axiom requires function $g$ to be total.

In addition, $T^0$ contains standard $\mathcal{BAT}$ axioms:

---

To see why, consider, for instance, a basic fluent $f$ declared as $f : \{0,1\} \rightarrow \{0,1\}$ and a dynamic causal law $\text{occurs}(a) \text{ causes } \text{dom}_f(1)$.

Intuitively, the axiom says that after $a$ is executed $f(1)$ must be defined, i.e., $f(1) = 0$ or $f(1) = 1$, which is non-deterministic.
State constraints for the basic fluents:

\[ dom_f(X) \text{ if } f(X) = Y \]
\[ dom_g(X) \text{ if } g(X) = Y \]

Definitions for the domains of attributes:

\[ dom_{attr_1}(X) \text{ if } attr_1(X) = Y \]
\[ dom_{attr_2}(X) \text{ if } attr_2(X) = Y \]

and the collection of axioms from (8).

### 2.3 Semantics of BATs

Intuitively, a basic action theory \( T \) defines the collection of discrete dynamic systems satisfying its axioms. The semantics of \( T \) will describe such systems by specifying their transition diagrams, often referred to as models of \( T \). Nodes of a transition diagram represent possible states of the dynamic system; arcs of the diagram are labeled by actions. A transition \( \langle \sigma_0, a, \sigma_1 \rangle \) says that the execution of action \( a \) in state \( \sigma_0 \) may take the system to state \( \sigma_1 \).

A state of the diagram will be defined by the universe — a collection of objects of the sorts of \( T \), and by a physically possible assignment of values to \( T \)'s functions. Moreover, we assume that the sorted universe and the values of statics and attributes are the same in all states, i.e., states only differ by the values of fluents.

To make this precise it is convenient to partition an interpretation \( I \) of an action signature \( \Sigma \) into two parts: fluent part consisting of the universe of \( I \) and the restriction of \( I \) on the sets of fluents, and static part consisting of the same universe and the restriction of \( I \) on the remaining elements of the signature. Sometimes we will refer to the latter as a static interpretation of \( \Sigma \).

We also need the following notation: Given an action signature \( \Sigma \) and a collection \( U \) of strings in some fixed alphabet, we denote by \( \Sigma_U \) the signature obtained from \( \Sigma \) by expanding its set of object constants by elements of \( U \), which we assume to be of sort universe.

**Definition 4 (Pre-model)**

Let \( T \) be a basic action theory with signature \( \Sigma \) and \( U \) be a collection of strings in some fixed alphabet. A static interpretation \( M \) of \( \Sigma_U \) is called a pre-model of \( T \) (with the universe \( U \)) if \( M(\text{universe}) = U \) and for every object constant \( o \) of \( \Sigma_U \) that is not an object constant of \( \Sigma \), \( M(o) = o \).

Given a pre-model \( M \) with the universe \( U \) we will often denote signature \( \Sigma_U \) by \( \Sigma_M \).

To illustrate this notion let us consider a pre-model of theory \( T \) from Example

**Example 2 (A pre-model of Basic Action Theory \( T^0 \))**

To define a pre-model of basic action theory \( T^0 \) from Example 1 let us consider a static interpretation \( M \) with the universe \( U_M = \{ x, y, z, a, b, true, false \} \) such that:
\( \mathcal{M}(\text{universe}) = \mathcal{M}(c_1) = \{x, y, z, a, b, \text{true, false}\}; \)
\( \mathcal{M}(c_2) = \{x\}; \)
\( \mathcal{M}(c_3) = \{y, z\}, \)
\( \mathcal{M}(\text{actions}) = \{a, b\}; \)
\( \mathcal{M}(a) = \{y\}; \) and
\( \mathcal{M}(\text{attr}_1)(a) = \mathcal{M}(\text{attr}_2)(b) = y. \)

In addition: every symbol from \( U_M \) is added to \( \Sigma^0_U \) and mapped into itself; \( \text{dom}_{\text{attr}_1} = \{a\}, \text{dom}_{\text{attr}_2} = \{b\} \); the interpretation of special function \( \text{link} \) is determined by the hierarchy from Figure 2; the interpretation of is\_a is extracted from the interpretation of the hierarchy’s sorts.

Clearly, \( \mathcal{M} \) satisfies the conditions in Definition 4 and hence is a pre-model of \( T^0. \)

A pre-model \( \mathcal{M} \) of \( T \) uniquely defines a model \( T_M \) of \( T \) if such a model exists. The definition of \( T_M \) will be given in two steps: first we define \( T_M \)'s states and then its transitions.

Intuitively, if theory \( T \) does not contain definitions, then a state of \( T_M \) is an interpretation \( I \) with static part \( \mathcal{M} \) that satisfies the state constraints of \( T. \) The situation is less simple for theories containing definitions (especially recursive ones). Similar to the case of \( \mathcal{AL} \), the definition of a state will be given using logic programs under the answer set semantics; specifically, we will use logic programs with non-Herbrand partial functions in the language \( \text{ASP}\{f\} \) \cite{Balduccini2013}.

Let \( \mathcal{M} \) be a pre-model of action theory \( T. \)

**Program \( S_M: \)**

By \( S_M \) we denote a logic program that consists of:

a) rules obtained from the state constraints and definitions of \( T \) by replacing variables with properly typed object constants of \( \Sigma_M, \) replacing object constants with their corresponding interpretations in \( \mathcal{M}, \) removing the constant \( \text{false} \) from the head of state constraints, and replacing the keyword \( \text{if} \) with \( \leftarrow, \)

b) the Closed World Assumption:

\[-d(t_0, \ldots, t_n) \leftarrow \neg d(t_0, \ldots, t_n)\]

for every defined function \( d : c_0 \times \ldots \times c_n \rightarrow \text{booleans} \) and \( t_i \in \mathcal{M}(c_i), \)

\( 0 \leq i \leq n. \)

**end of \( S_M: \)**

Finally, we define a program \( S_I \) used in the definition of states of the transition diagram defined by \( \mathcal{M}. \)

**Program \( S_I: \)**

\(^3\) Other approaches for introducing non-Herbrand functions in ASP can be seen, for instance, in \( \text{Cabalar 2011; Lifschitz 2012; Bartholomew and Lee 2013.} \)
For every interpretation $\mathcal{I}$ of $\Sigma$ with static part $\mathcal{M}$, by $S_\mathcal{I}$ we denote the logic program obtained by adding to $S_\mathcal{M}$ the set of atoms obtained from $\mathcal{I}$ by removing the defined atoms.

**end of $S_\mathcal{I}$**

**Definition 5 (State)**

Let $\mathcal{M}$ be a pre-model of a BAT theory $T$. An interpretation $\sigma$ with static part $\mathcal{M}$ is a state of the transition diagram $T_\mathcal{M}$ defined by $\mathcal{M}$ if $\sigma$ is the only answer set of $S_\sigma$.

Notice that $\sigma$ is not a state if $S_\sigma$ has multiple answer sets, a situation that would only occur when the value of some defined function is not completely determined by the values of basic functions. We will return to this issue later, in Section 4.2.

**Example 3 (States of the diagram)**

Let $\mathcal{M}$ be the pre-model of theory $T^0$ from Example 2. The program $S_\mathcal{M}$ for this $\mathcal{M}$ looks as follows:

\[
\begin{align*}
\leftarrow & \neg \text{dom}_a(x), \text{instance}(x, c_2). \\
\text{dom}_f(x) & \leftarrow f(x) = y \\
\text{dom}_f(x) & \leftarrow f(x) = z \\
\text{dom}_g(x) & \leftarrow g(x) = y \\
\text{dom}_g(x) & \leftarrow g(x) = z \\
\text{dom}_{\text{attr}_1}(a) & \leftarrow \text{attr}_1(a) = y \\
\text{dom}_{\text{attr}_1}(a) & \leftarrow \text{attr}_1(a) = z \\
\text{dom}_{\text{attr}_2}(a) & \leftarrow \text{attr}_2(a) = y \\
\text{dom}_{\text{attr}_2}(a) & \leftarrow \text{attr}_2(a) = z \\
\text{dom}_{\text{attr}_1}(b) & \leftarrow \text{attr}_1(b) = y \\
\text{dom}_{\text{attr}_1}(b) & \leftarrow \text{attr}_1(b) = z \\
\text{dom}_{\text{attr}_2}(b) & \leftarrow \text{attr}_2(b) = y \\
\text{dom}_{\text{attr}_2}(b) & \leftarrow \text{attr}_2(b) = z
\end{align*}
\]

and the Closed World Assumptions for the special functions. Recall that, according to the definition of an interpretation of a sorted signature, for every $x \in |\mathcal{I}|$, $\mathcal{I}(\text{is}_a(x, c))$ is true iff $c$ is a source node of the sort hierarchy and $\mathcal{I}(x) \in \mathcal{I}(c)$, and for every object $o$ and sort $c$, $\mathcal{I}(\text{is}_a)(\mathcal{I}(o), c)$ is true iff $\langle o, c \rangle$ is a link in our hierarchy. This, together with the condition on the interpretation of link guarantees that every state of $T_\mathcal{M}$ contains atoms $\text{is}_a(x, c_2)$, $\text{is}_a(y, c_3)$, and other atoms formed by $\text{is}_a$ and $\text{link}$ that define our hierarchy. The collection of these atoms together with the closed world assumptions for $\text{is}_a$, $\text{link}$ and the other defined statics uniquely determine their values. It is easy to check that every state of $\mathcal{M}$ contains literals formed by these special fluents. Every state of $T_\mathcal{M}$ also contains $\text{attr}_1(a) = y$, $\text{attr}_2(b) = y$, and $\text{dom}_g(x)$. Overall, $T_\mathcal{M}$ has the following six states (for each state, we only show non-special fluents):

\[
\begin{align*}
\sigma_1 &= \{ f(x) = y, g(x) = y \} \\
\sigma_2 &= \{ f(x) = z, g(x) = y \} \\
\sigma_3 &= \{ f(x) = y, g(x) = z \} \\
\sigma_4 &= \{ f(x) = z, g(x) = z \} \\
\sigma_5 &= \{ g(x) = y \} \\
\sigma_6 &= \{ g(x) = z \}.
\end{align*}
\]
In addition, states $\sigma_1, \sigma_2, \sigma_3,$ and $\sigma_4$ contain $\text{dom}_f(x)$ while states $\sigma_5$ and $\sigma_6$, in which $f$ is undefined on $x$, contain $\neg\text{dom}_f(x)$.

To define transitions of the diagram that corresponds to a pre-model $M$ with the universe $U$, we construct a logic program $P_M$ whose signature is obtained from the signature of program $S_M$ defined above by

- adding a new sort, step, ranging over 0 and 1;
- replacing every fluent $f : c_0 \times \ldots \times c_n \rightarrow c$ by function $f : c_0 \times \ldots \times c_n \times \text{step} \rightarrow c$;
- adding a function symbol $\text{occurs} : \text{actions} \times \text{step} \rightarrow \text{booleans}$.

**Program $P_M$:**

Program $P_M$ is obtained from a theory $T$ and pre-model $M$ by

a) replacing variables by properly typed object constants of $\Sigma_M$;
b) replacing object constants by their corresponding interpretations in $M$;
c) removing the object constant false from the head of state constraints;
d) replacing every occurrence of a fluent term $f(\overline{t})$ in the head of a dynamic causal law by $f(\overline{t}, I + 1)$;
e) replacing every other occurrence of a fluent term $f(\overline{t})$ by $f(\overline{t}, I)$;
f) removing “occurs(a) causes” from every dynamic causal law and adding \text{occurs}(a) to the body;
g) replacing “impossible \text{occurs}(a)” in every executability condition by $\neg\text{occurs}(a)$;
h) replacing $\text{occurs}(a)$ by $\text{occurs}(a, I)$ and $\neg\text{occurs}(a)$ by $\neg\text{occurs}(a, I)$;
i) replacing the keyword if by $\Leftarrow$;
j) adding the Closed World Assumption:
\[
\neg d(t_0, \ldots, t_n, I) \Leftarrow \neg d(t_0, \ldots, t_n, I)
\]
for every defined fluent $d : c_0 \times \ldots \times c_n \rightarrow \text{booleans}$ and $t_i \in M(c_i)$, $0 \leq i \leq n$;
k) adding the rule:
\[
\neg f(t_0, \ldots, t_n) \Leftarrow \neg f(t_0, \ldots, t_n)
\]
for every defined static of the form $f : c_0 \times \ldots \times c_n \rightarrow \text{booleans}$ and $t_i \in M(c_i)$, $0 \leq i \leq n$;
l) adding the Inertia Axiom:
\[
\text{dom}_f(t_0, \ldots, t_n, I + 1) \Leftarrow \text{dom}_f(t_0, \ldots, t_n, I),
\]
\[
\neg \text{dom}_f(t_0, \ldots, t_n, I + 1) \Leftarrow \neg \text{dom}_f(t_0, \ldots, t_n, I),
\]
for every basic fluent $\text{dom}_f : c_0 \times \ldots \times c_n \rightarrow \text{booleans}$, and $t_i \in M(c_i)$, $0 \leq i \leq n$;
m) adding the Inertia Axiom:
\[
f(t_0, \ldots, t_n, I + 1) = t \Leftarrow \text{dom}_f(t_0, \ldots, t_n, I + 1),
\]
\[
f(t_0, \ldots, t_n, I) = t, \quad \neg f(t_0, \ldots, t_n, I + 1) \neq t
\]
for every basic fluent \( f : c_0 \times \ldots \times c_n \rightarrow c \) not formed by \( \text{dom} \), and \( t_i \in M(c_i), 0 \leq i \leq n \), and \( t \in M(c) \).

end of \( P_M \)

Note that the last axiom is a modification of the standard logic programming version of the Inertia Axiom (see, for instance, (Gelfond and Kahl 2014)), which is stated for total (boolean) functions. The main difference is the addition of the domain statements in the body. The inertia axiom for the function \( \text{dom} \) is of the standard form.

**Program** \( P(M, \sigma_0, a) \): Let \( \sigma_0 \) be a state of the transition diagram defined by a pre-model \( M \), and let \( a \subseteq M(\text{actions}) \). By \( P(M, \sigma_0, a) \) we denote the logic program formed by adding to \( P_M \) the set of atoms obtained from \( \sigma_0 \) by replacing every fluent atom \( f(t_0, \ldots, t_n) = t \) by \( f(t_0, \ldots, t_n, 0) = t \) and adding the set of atoms \( \{ \text{occurs}(x, 0) : x \in a \} \).

end of \( P(M, \sigma_0, a) \)

**Definition 6 (Transition)**
Let \( \sigma_0 \) and \( \sigma_1 \) be states of the transition diagram defined by a pre-model \( M \) and let \( a \subseteq M(\text{actions}) \). The triple \( \langle \sigma_0, a, \sigma_1 \rangle \) is a transition of the transition diagram defined by a pre-model \( M \) of a \( \text{BAT} \) theory \( T \) if program \( P(M, \sigma_0, a) \) has an answer set \( A \) such that \( f(t_0, \ldots, t_n) = t \in \sigma_1 \) iff

- \( f \) is an attribute or a static and \( f(t_0, \ldots, t_n) = t \in A \), or
- \( f \) is a fluent and \( f(t_0, \ldots, t_n, 1) = t \in A \).

**Definition 7 (Model)**
A transition diagram \( T_M \) defined by a pre-model \( M \) of a basic action theory \( T \) is called a model of \( T \) if it has a non-empty collection of states.

The following example illustrates the definition.

**Example 4 (A Model of Basic Action Theory \( T^0 \))**
To define a model of theory \( T^0 \) from Example 1 let us consider the pre-model \( M \) from Example 2. States of the diagram defined by this pre-model were given in Example 3. To define the transitions of the model defined by \( M \) we use Definition 6. Let us illustrate this by showing that a triple \( \langle \sigma_1, b, \sigma_5 \rangle \) is a transition. To do that we need first to construct a program \( P(M, \sigma_1, b) \) (we are only showing rules relevant to our argument):

\[
\begin{align*}
[1] & \quad f(x, 1) = y \leftarrow \text{instance}(b, \text{actions}), \\
& \quad \text{occurs}(b, 0), \\
& \quad \text{attr}_1(b) = y, \\
& \quad g(x, 0) = y.
\end{align*}
\]

\[
\begin{align*}
[2] & \quad \neg \text{dom}(x, 1) \leftarrow \text{instance}(b, \text{actions}), \\
& \quad \text{occurs}(b, 0), \\
& \quad \text{attr}_2(b) = y.
\end{align*}
\]
Daniela Inclezan and Michael Gelfond

\[ \text{dom}(x, 0) \leftarrow f(x, 0) = y. \]
\[ \text{dom}(x, 1) \leftarrow f(x, 1) = y. \]
\[ \text{dom}_g(x, 0) \leftarrow g(x, 0) = y. \]
\[ \text{dom}_g(x, 1) \leftarrow g(x, 1) = y. \]

\[ \text{dom}(x, 1) = y \leftarrow \text{dom}(x, 1), \]
\[ f(x, 0) = y, \]
\[ \text{not } f(x, 1) \neq y. \]

\[ g(x, 1) = y \leftarrow \text{dom}_g(x, 1), \]
\[ g(x, 0) = y, \]
\[ \text{not } g(x, 1) \neq y. \]

\[ \text{dom}(x, 1) \leftarrow \text{dom}(x, 0), \]
\[ \text{not } \neg \text{dom}(x, 1). \]

\[ \text{dom}_g(x, 1) \leftarrow \text{dom}_g(x, 0), \]
\[ \text{not } \neg \text{dom}_g(x, 1). \]

\[ f(x, 0) = y. \]
\[ g(x, 0) = y. \]
\[ \text{occurs}(b, 0). \]

It is easy to see that the program has a unique answer set, say, \( S \). Since \( \sigma_5 = \{ g(x) = y \} \) we need to show that the only fluent atom with the step parameter 1 belonging to \( S \) is \( g(x, 1) = y \). By the second rule from group [5], \( \text{dom}_g(x, 1) \in S \).

By the second rule of [4] we have that \( g(x, 1) = y \in S \). As expected, function \( g \) maintains its value by inertia. The situation is different for \( f \). By rule [2] we have that \( \neg \text{dom}_f(x, 1) \in S \) and hence neither rule [5] nor [4] for \( f \) are applicable. Rule [1] is also not applicable since \( \text{attr}_1 \) is not defined for \( b \). Therefore the state defined by \( S \) is exactly \( \sigma_5 = \{ g(x) = y \} \). (Note that the argument would not be possible if we were to use the traditional version of the Inertia Axiom. The modification related to the treatment of \( \text{dom} \) presented in axioms [4] and [5] is essential.)

Using the same method one can easily verify that triples \( \langle \sigma_2, a, \sigma_1 \rangle \), \( \langle \sigma_5, a, \sigma_1 \rangle \), \( \langle \sigma_5, b, \sigma_5 \rangle \), etc. are transitions of the transition diagram defined by \( M \).

### 2.4 Entailment Relation

Let us consider a fixed action theory \( T \) with action signature \( \Sigma \), and define an entailment relation between \( T \) and statements of \( \Sigma \).

Let \( I \) be an interpretation of \( \Sigma \). A ground instance of a statement \( \alpha \) of \( \Sigma \) with respect to \( I \) is a statement obtained by replacing variables of \( \alpha \) by properly typed object constants in \( \Sigma_I \) and replacing object constants of \( \alpha \) by their interpretations in \( I \).

Now let us consider a model \( T_M \) of a basic action theory \( T \) defined by a pre-model \( M \) with the universe \( U \) and let \( \sigma \) be a state of \( T_M \).
Modular Action Language ALM

Definition 8 (Satisfiability Relation for Ground Statements of a BAT)

- A state $\sigma$ of $T_M$ satisfies a ground state constraint $\alpha$ if $\sigma$ contains the head of $\alpha$ whenever it contains its body.
- A state $\sigma$ of $T_M$ satisfies a ground definition $\alpha$ if $\sigma$ contains the head of a clause in $\alpha$ if $\sigma$ contains a clause with the same head and the body belonging to $\sigma$.
- A transition $\langle \sigma_0, a, \sigma_1 \rangle$ of $T_M$ satisfies a ground dynamic causal law $\alpha$ that starts with the expression “occurs($e$) causes” if $a$ contains action $e$ and $\sigma_1$ contains the head of $\alpha$ whenever $\sigma_0$ contains its body.
- A transition $\langle \sigma_0, a, \sigma_1 \rangle$ of $T_M$ satisfies a ground executability condition $\alpha$ that starts with the expression “impossible $e$” if either (1) $a$ does not contain $e$ or (2) the body of $\alpha$ contains:
  - a ground literal $l$ such that $l \notin \sigma_0$, or
  - an expression “occurs($e_1$)” such that $e_1 \notin a$, or
  - an expression “¬occurs($e_2$)” such that $e_2 \in a$.

Definition 9 (Satisfiability Relation for Arbitrary Statements of a BAT)

Let $T_M$ be a model of a basic action theory $T$ defined by a pre-model $M$ with the universe $U$.

- $T_M$ satisfies a constraint $\alpha$ over signature $\Sigma$ of $T$ if every state of $T_M$ satisfies all ground instances of $\alpha$ with respect to $U$. Similarly for definitions.
- $T_M$ satisfies a dynamic causal law $\alpha$ over signature $\Sigma$ of $T$ if every transition of $T_M$ satisfies all ground instances of $\alpha$ with respect to $U$. Similarly for executability conditions.

Definition 10 (Entailment)

A statement $\alpha$ is entailed by a theory $T$ ($T \models \alpha$) if $\alpha$ is true in every model of $T$.

Having the notion of entailment allows us to investigate the relationship between causal laws. For instance we can show that

\[
\{\text{occurs}(A) \text{ causes } f \text{ if } p, q; \text{ occurs}(A) \text{ causes } f \text{ if } \neg p\} \models \text{occurs}(A) \text{ causes } f \text{ if } q
\]

\[
\{\text{occurs}(A) \text{ causes } f \text{ if } p, q; \quad q \text{ if } p\} \models \text{occurs}(A) \text{ causes } f \text{ if } p
\]

etc. Our notion of entailment is somewhat similar to the notion of subsumption from (Eiter et al. 2010) - a relation between an action description and a query (including queries having the form of causal laws and executability conditions). Our entailment relation can be viewed as a generalization of subsumption from system descriptions to theories. It allows variables and, unlike that of subsumption, is defined in terms of multiple transition diagrams specified by the theory. There are also related formalisms that allow entailment of causal laws and executability conditions (see, for instance (Turner 1999) and (Giunchiglia et al. 2004)). There are many interesting problems related to the ALM entailment, including that of finding a sound and complete set of inference rules for it. We hope to address these problems in our future work.
In this section we use examples to introduce the syntax of theories and system descriptions of $\mathcal{ALM}$ and define their semantics. (The full grammar for the language can be seen in Appendix A.)

We begin with describing unimodule system descriptions, i.e. system descriptions whose theories consist of exactly one module.

### 3.1 Unimodule System Descriptions

We start with a comparatively simple problem of formalizing the domain described by the following story:

**Example 5 (A Travel Domain)**

Consider a travel domain in which there are two agents, Bob and John, and three locations, New York, Paris, and Rome. Bob and John can move from one location to another if the locations are connected.

If we were to represent this knowledge in $\mathcal{AL}$ we would start with identifying objects of the domain including actions such as, say, $go(bob, paris, rome)$ and write $\mathcal{AL}$ axioms describing the relationships between these objects. The use of $\mathcal{ALM}$ suggests a very different methodology.

**Methodology of Describing Dynamic Domains in $\mathcal{ALM}$:**

1. Determine what sorts of objects are relevant to the domain of discourse and how these sorts can be organized into an inheritance hierarchy.
2. Use $\mathcal{ALM}$ to describe the basic action theory for this type of domains. This should be done in two steps:
   - Describe the action signature of our abstraction by declaring sorts (together with their attributes and the inheritance hierarchy), basic and defined statics and fluents. (Notice that this signature normally will not contain particular objects of our story. It would have no mention of Bob, Paris, etc. However, the signature may include some object constants pertinent to the general domain of the story – see for instance the Monkey and Banana Problem in Section 4.1)
   - Use this action signature to formulate axioms of the theory.
3. Populate sorts of your hierarchy with objects relevant to your story and describe these objects and their sort membership in $\mathcal{ALM}$.

As is the case with other problem solving methodologies, we begin by choosing a proper level of abstraction for our example. Since the example is used for illustrative purposes we opted for using the following simple abstraction:

Our domains will contain things and discrete points in space. Certain things, called agents, will be able to move from one point to another if the two points are connected. We are interested in the relations between points and the locations of things, including changes of these locations caused by a sequence of given moves.
(Note that our abstraction does not allow a location to be a part of another location, e.g., we will not be able to express that Paris is located in France. It ignores the means of transportation, the possibility that locations may have restrictions on the number of things they can contain, etc.)

Accordingly, our basic action theory containing commonsense knowledge about motion formulated in these terms will include sorts \textit{things}, \textit{agents}, \textit{points}, and \textit{move}, together with special sorts \textit{universe} and \textit{actions}, which belong to every action signature.

We call this basic action theory $T_{bm}$. The sorts of $T_{bm}$ will be organized in a hierarchy depicted in Figure 2.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{sort_hierarchy}
\caption{Sort Hierarchy for $T_{bm}$}
\end{figure}

Our next step is to describe $T_{bm}$ in ALM.

\textbf{Example 6 (Motion Theory in ALM)}

The description of a theory in ALM starts with the keyword \texttt{theory} and is followed by a collection of modules. Our theory, called \texttt{basic\_motion}, consists of only one module \texttt{moving}

\begin{verbatim}
theory basic\_motion
  module moving
    (module body)
\end{verbatim}

where \texttt{(module body)} stands for the declarations of sorts, functions, and axioms of the theory. We assume that \textit{things}, \textit{points}, and \textit{agents} have no attributes, while actions from the sort \textit{move} may come with attribute \textit{actor} indicating the mover involved in the action, and attributes \textit{origin} and \textit{destination} (abbreviated as \textit{dest}) describing the locations of the actor before and after the execution of the action. Syntactically, all this information is specified as:

\begin{verbatim}
sort declarations
  points, things :: universe
  agents :: things
\end{verbatim}
move :: actions
  attributes
  actor : agents
  origin : points
  dest : points

The construct :: is called specialization and corresponds to the links of the sort hierarchy; for instance, the link from agents to things in Figure 2 is recorded by the statement agents :: things. Multiple links going into the same sort can be recorded by a single statement, as in points, things :: universe. Note that the special sorts universe and actions do not have to be declared. In case of a sort hierarchy with multiple links from $c$ to $pc_1, \ldots, pc_k$ we will use a specialization statement of the form $c :: pc_1, \ldots, pc_k$. In describing the attributes of actions of the sort move we use a shorthand. Attributes of move are functions defined on elements of the sort move, which means that the definition of, say, attribute actor should be written as actor : move $\to$ agents. After some deliberation however, we decided to allow to write it simply as actor : agents. The same agreement holds for attributes with a larger number of parameters; an attribute of a sort $c$ that has the form $\text{attr name} : c \times c_0 \times \ldots \times c_n \to c_{n+1}$ can be written as $\text{attr name} : c_0 \times \ldots \times c_n \to c_{n+1}$. This completes the description of the syntactic representation of our sort hierarchy in ALM.

The next step is to syntactically describe functions in the signature. One of the functions mentioned in our informal description specifies whether two points are connected or not. Let us call it connected. In general, the value of connected can be changed by actions (airports can be closed, roads blocked, etc.) and hence we define connected to be a basic fluent. In some scenarios, the property connected will be a symmetric relation but not in others; similarly, it may be a transitive relation or not. To allow for elaboration tolerance, we introduce two basic static functions, symmetric_connectivity and transitive_connectivity to characterize the property connected. The other function relevant to our domain maps things into points at which they are located. Let us call it loc_in. The value of the function can be changed by actions of our domain, hence it is a fluent. It is not defined in terms of other functions, thus it is a basic fluent. It is also a total function, as we assume that the location of every thing is defined in every state. In ALM these functions are syntactically declared as:

```plaintext
function declarations
  statics
    basic
      symmetric_connectivity : booleans
      transitive_connectivity : booleans
  fluents
    basic
      connected : points $\times$ points $\to$ booleans
      total loc_in : things $\to$ points
```
In this example the keywords **function declarations** are followed by the lists of statics and fluents. Elements from each list are divided into basic and defined with each total function in the list preceded by the keyword **total**. Naturally, the declaration of a sort, static, or fluent in a module should be unique.

This concludes our description of action signature of $T_{bm}$.

Now we are ready to define the collection of axioms of $T_{bm}$. In $\mathcal{ALM}$, we precede this collection by the keyword **axioms**. Each axiom will be ended by a period (.), as in:

```latex
axioms
\begin{align*}
\text{occurs}(X) & \text{ causes } \text{loc\_in}(A) = D & \text{if } & \text{instance}(X, \text{move}), \\
& & & \text{actor}(X) = A, \\
& & & \text{dest}(X) = D.
\end{align*}
```

\begin{align*}
\text{connected}(X, X). \\
\text{connected}(X, Y) & \text{ if } \text{connected}(Y, X), \\
& \text{symmetric\_connectivity}. \\
\neg\text{connected}(X, Y) & \text{ if } \\
& \neg\text{connected}(Y, X), \\
& \text{symmetric\_connectivity}. \\
\text{connected}(X, Z) & \text{ if } \text{connected}(X, Y), \\
& \text{connected}(Y, Z), \\
& \text{transitive\_connectivity}.
\end{align*}

\begin{align*}
\text{impossible} \text{ occurs}(X) & \text{ if } \text{instance}(X, \text{move}), \\
& \text{actor}(X) = A, \\
& \text{loc\_in}(A) \neq \text{origin}(X).
\end{align*}

\begin{align*}
\text{impossible} \text{ occurs}(X) & \text{ if } \text{instance}(X, \text{move}), \\
& \text{actor}(X) = A, \\
& \text{loc\_in}(A) = \text{dest}(X).
\end{align*}

\begin{align*}
\text{impossible} \text{ occurs}(X) & \text{ if } \text{instance}(X, \text{move}), \\
& \text{actor}(X) = A, \\
& \text{loc\_in}(A) = O, \\
& \text{dest}(X) = D, \\
& \neg\text{connected}(O, D).
\end{align*}

The keyword **total** in the declaration of the basic fluent $\text{loc\_in}$ stands for the axiom

\begin{align*}
\text{false} & \text{ if } \neg\text{dom}_{\text{loc\_in}}(X).
\end{align*}

that would otherwise have to be included among the axioms above. In general, the keyword **total** included in the declaration of a function $f : c_0 \times \ldots \times c_n \rightarrow c$ stands for the axiom

\begin{align*}
\text{false} & \text{ if } \neg\text{dom}_f(X_0, \ldots, X_n).
\end{align*}
This completes our description of the basic action theory $T_{bm}$ in $\mathcal{ALM}$.

Note that the semantics of the unimodule $\mathcal{ALM}$ theory $\text{basic\_motion}$ is given by the basic action theory $T_{bm}$ defined by it. In the following sections we will present other examples of basic action theories and their interpretations represented in $\mathcal{ALM}$. (Whenever possible we will make no distinction between these theories and their $\mathcal{ALM}$ representations.)

As discussed above, a basic action theory $T$ is used to define the collection of its models — transition diagrams representing dynamic domains with shared ontology and properties. Usually, a knowledge engineer is interested in one such domain, characterized by particular objects, sorts, and values of statics. If the engineer’s knowledge about this domain is complete, the domain will be represented by a unique model of $T$. Otherwise there can be several alternative models.

The syntactic construct of $\mathcal{ALM}$ used to define such knowledge is called a structure and has the form

```
structure name
  ⟨structure body⟩
```

where $⟨structure body⟩$ stands for the definition of objects in the hierarchy of $T_{bm}$ and the values of its statics. Let us illustrate the use of this construct by the following example:

Example 7 ($\mathcal{ALM}$’s Representation of a Specific Basic Motion Domain.)

Let us consider the $\mathcal{ALM}$ theory $\text{basic\_motion}$ from Example 6, which encodes the basic action theory $T_{bm}$, and use $\mathcal{ALM}$ to specify the particular basic motion domain from Example 5.

The $\mathcal{ALM}$ definition of the structure used to describe this domain starts with the header:

```
structure Bob_and_John
```

followed by the definition of agents and points:

```
instances
  bob, john in agents
  new_york, paris, rome in points
```

To specify particular actions of our domain we expand our list of instances by

```
go(X, P_1, P_2) in move where P_1 \neq P_2
  actor = X
  origin = P_1
  dest = P_2
```

Note that the last definition describes several instances simultaneously via the use of variables; we call this type of definition an instance schema. The instance schema defining $go(X, P_1, P_2)$ stands for the collection of instance definitions:
The condition \( P_1 \neq P_2 \) ensures that Bob and John do not move to a destination identical to the origin.

The following would also be a valid instance schema:

\[
\text{go}(X, P) \text{ in move} \\
\text{actor} = X \\
\text{dest} = P
\]

if we were interested only in the destinations of Bob and John’s movements, but not in their origins.

In our example connectivity between points is both symmetric and transitive: This is captured syntactically by the following:

\[
\text{values of statics} \\
\text{symmetric_connectivity,} \\
\text{transitive_connectivity.}
\]

This concludes our definition of Bob_and_John structure.

To syntactically relate a theory with its structure, we use the construct of ALM called system description. In our case it will look as follows:

\[
\text{system description travel} \\
\text{theory basic_motion} \\
\text{module moving} \\
\text{(module body)} \\
\text{structure Bob_and_John} \\
\text{(structure body)}
\]

where \((\text{module body})\) and \((\text{structure body})\) are defined in Examples 6 and 7. The system description travel contains all the information we considered relevant to our particular travel domain. It is not difficult to see that this knowledge is complete and therefore describes exactly one model (i.e., one transition diagram) of basic_motion. This is exactly the model we intended for our domain. A part of this model can be seen in Figure 3. We only show fluent loc_in and assume that in every state of the

---

5 If a theory contains an object constant \( o \) then its value, say \( y \), can be declared as:

\[
\text{object constants} \\
o = y
\]

If the structure contains no assignment of value to constant \( o \), we assume that \( o \) belongs to the structure’s universe and is mapped into itself.
part of the diagram shown in the picture Paris and Rome are connected to each other, but neither of them is connected to New York; we use shorthands $b$, $j$, $ny$, $p$, and $r$ for bob, john, new_york, paris, and rome respectively; and we only show arcs that are labeled by a single action.

Fig. 3. (Partial) Transition Diagram for System Description travel

The model is unique because we specified the membership of our objects in the source nodes of the hierarchy. This information is sufficient to uniquely define the universe and the interpretations of all the sorts.

The next example illustrates how incomplete information about a domain can lead to multiple models of the system description of this domain:

Example 8 (System Description with Multiple Models)
Consider a system description underspecified_hierarchy consisting of a theory professors and a structure alice:

system_description underspecified_hierarchy
theory professors
module professors
sort declarations
   professor :: person
   assistant, associate, full :: professor
axioms
   false if instance(X, $C_1$), instance(X, $C_2$),
   link($C_1$, professor), link($C_2$, professor),
   $C_1 \neq C_2$. 


structure alice
instances
alice in professor

The theory describes a simple hierarchy. The structure populates the hierarchy with one member, Alice (see Figure 4). Unfortunately all we know about Alice is that she is a professor. It is not difficult to check that this system description has three models. In the first one Alice is an assistant professor, in the second she is an associate professor, and in the third one - a full professor.

Fig. 4. Underspecified Hierarchy

We hope that these examples gave the reader a sufficient insight in the meaning of unimodule $\mathcal{ALM}$ theories and system descriptions. In general, the semantics of a syntactically correct unimodule theory $\mathcal{T}$ of $\mathcal{ALM}$ is given by the unique BAT defined by $\mathcal{T}$. Similarly, the semantics of a system description $\mathcal{D}$ of $\mathcal{ALM}$ is given by models of the BAT theory defined by $\mathcal{T}$ and by the set of interpretation defined by the structure of $\mathcal{D}$.

3.2 Organizing Knowledge into Modules

So far we only considered very simple $\mathcal{ALM}$ theories consisting of one module. To create theories containing a larger body of knowledge we need multiple modules organized into a module hierarchy. To illustrate this concept let us consider an extension of basic action theory $T_{bm}$ of motion by an additional sort of things called carriables, which can be carried between connected points by agents that are holding them. Recall from Example 6 that we represented the original $T_{bm}$ as an $\mathcal{ALM}$ theory called basic motion, with a unique module moving. We will use the name motion for the $\mathcal{ALM}$ theory that will specify the extension of $T_{bm}$. The new theory will contain the moving module developed above as well as a new module called carrying things:

theory motion
module moving
  ⟨module body⟩
module carrying_things
  ⟨module body⟩

In addition to sorts, fluents, and axioms from module moving, the signature of the new module carrying_things will contain two new sorts, carriables and carry; a new inertial fluent, holding; and a defined fluent, is_held. Informally, holding will be understood as having in one’s hands and carry as moving while holding, which will allow us to define carry as a special case of move.
The dependency of \texttt{carrying\_things} on \texttt{moving} is expressed in ALM by the syntactic construct \texttt{depends on} called \textit{module dependency} as follows:

\begin{verbatim}
module carrying\_things
  depends on moving
\end{verbatim}

This says that the sorts and functions explicitly declared in \texttt{carrying\_things} depend on sorts and functions declared in the module \texttt{moving}. We say that the declarations of \texttt{moving} are implicit in module \texttt{carrying\_things}. We require all sorts and functions appearing in a module to be either explicitly or implicitly declared in that module. By means of the module dependency construct, a theory of ALM can be structured into a hierarchy of modules. The dependency relation of this hierarchy should form a DAG. Now we define the body of the new module:

\begin{verbatim}
sort declarations
  carriables :: things
carry :: move
attributes
  carried\_object : carriables
\end{verbatim}

Note that, since \texttt{carry} is defined as a special case of \texttt{move}, it automatically inherits the attributes of \texttt{move}; hence those attributes do not have to be repeated in the declaration of \texttt{carry}. Next, the module contains the declarations of functions:

\begin{verbatim}
function declarations
  fluents
    basic
      total holding : agents \times things \rightarrow booleans
    defined
      is\_held : things \rightarrow booleans
\end{verbatim}

and the new axioms:

\begin{verbatim}
axioms
  loc\_in(C) = P \text{ if } holding(T, C),
  loc\_in(T) = P \text{ if } holding(T, C),
  is\_held(X) \text{ if } holding(T, X).
  \text{impossible occurs}(X) \text{ if } instance(X, move),
  actor(X) = A,
  is\_held(A).
  \text{impossible occurs}(X) \text{ if } instance(X, carry),
  actor(X) = A,
  carried\_object(X) = C,
  \neg holding(A, C).
\end{verbatim}

The first two axioms say that an agent and an object he is holding have the same location. The next defines fluent \texttt{is\_held(X)} – object \texttt{X} is held by someone or something. The first executability condition states that to move an actor should be
free (i.e., not held). The second states that it is impossible to carry a thing without holding it.

Structuring a theory of $ALM$ into a hierarchy of modules has several advantages. First, this supports the stepwise development of a knowledge base by allowing parts of its theory to be developed and tested independently from other parts. Second, it increases the readability of $ALM$ theories, due to the more manageable size of their modules. And finally, this approach facilitates the creation of knowledge libraries. Theories containing very general information can be stored in a library and imported from there when constructing system descriptions. For instance, imagine that our motion theory is stored in a library called commonsense_library. The system description travel could then be re-written by importing this theory as follows:

```
   system description travel
import theory motion from commonsense_library
   structure Bob_and_John
   (structure body)
```

We hope that these examples gave the reader some insight into the meaning of theories of $ALM$ that have more than one module. The accurate semantics for such a theory $T$ is given by its flattening, i.e., by translating $T$ into the unimodular theory with the same intuitive meaning.

First, we will give the semantics of theories satisfying the semantic conditions given in the following definition, theories that we call semantically coherent.

**Definition 11 (Semantically Coherent Theory)**
A theory of $ALM$ is semantically coherent if it satisfies the following conditions:

- All sorts and functions appearing in a module of $T$ are (explicitly or implicitly) declared in that module.
- The module hierarchy of $T$ defined by relation “depends on” forms a DAG, $G$. (The nodes of $G$ correspond to modules of $T$. An arc $\langle M_2, M_1 \rangle$ is in $G$ if and only if module $M_2$ contains the statement “depends on $M_1$”.)
- No two modules of a theory contain different declarations of the same sort or the same function name.

The last condition in Definition 11 can be weakened to allow the use of the same name for a function and its restriction on a smaller sort. This and other similar features however can somewhat distract from the main ideas of $ALM$ and will not be included in the original version of $ALM$.

The flattening $f(T)$ of an $ALM$ theory $T$ is constructed by the following algorithm:

1. Select modules $M_1$ and $M_2$ of $T$ such that $M_1$ contains the statement “depends on $M_2$”.

---

6 For greater readability, we recommend maintaining a balance between a manageable module size and a relatively shallow module dependence hierarchy.
2. Replace $M_1$ and $M_2$ by the new module $M$ obtained by uniting depends on statements, sort declarations, object constant declarations, function declarations, and axioms of $M_2$ with those of $M_1$.
3. Remove the statement “depends on $M_2$” from $M$.
4. Replace $M_1$ and $M_2$ in all the statements of $T$ of the form “depends on $M_1$” and “depends on $M_2$” by $M$.
5. Repeat until no dependent modules exist.
6. Construct a new module with declarations and axioms defined as unions of the corresponding declarations and axioms of the remaining modules.
7. Return the resulting unimodule theory $f(T)$.

The second condition in Definition 11 guarantees that the algorithm will terminate. The first and second conditions ensure that the result of the algorithm does not contain the depends on statement and that all sorts and functions within module $M$ of step 2 have unique (explicit or implicit) declarations. Thanks to condition three this property is preserved by step 6 of the algorithm and hence $f(T)$ is indeed a unimodule theory.

As expected, the semantics of an ALM theory $T$ with more than one module is given by the semantics of the unimodule theory $f(T)$.

For illustrative purposes we give the result of applying the flattening algorithm to the motion theory given above:

```plaintext
theory flat_motion
module flat_motion

sort declarations
    points, things :: universe
    agents, carriables :: things
    move :: actions
attributes
    actor : agents
    origin : points
    dest : points
carry :: move
attributes
    carried_object : carriables

function declarations
statics
    basic
        symmetric_connectivity : booleans
        transitive_connectivity : booleans
fluenets
    basic
        total loc_in : things → points
        total holding : agents × things → booleans
        connected : points × points → booleans
```

defined

\( \text{is\_held} : \text{things} \rightarrow \text{booleans} \)

axioms

\( \text{occurs}(X) \text{ causes } \text{loc\_in}(A) = D \text{ if } \text{instance}(X, \text{move}), \text{actor}(X) = A, \text{dest}(X) = D. \)

\( \text{connected}(X, X). \) if \( \text{connected}(Y, X), \text{symmetric\_connectivity}. \)

\( \neg\text{connected}(X, Y) \) if \( \neg\text{connected}(Y, X), \text{symmetric\_connectivity}. \)

\( \text{connected}(X, Z) \) if \( \text{connected}(X, Y), \text{connected}(Y, Z), \text{transitive\_connectivity}. \)

\( \text{loc\_in}(C) = P \text{ if } \text{holding}(T, C), \text{loc\_in}(T) = P. \)

\( \text{loc\_in}(T) = P \text{ if } \text{holding}(T, C), \text{loc\_in}(C) = P. \)

\( \text{is\_held}(C) \text{ if } \text{holding}(T, C). \)

\( \text{impossible} \text{ occurs}(X) \text{ if } \text{instance}(X, \text{move}), \text{actor}(X) = A, \text{origin}(X) \neq \text{loc\_in}(A). \)

\( \text{impossible} \text{ occurs}(X) \text{ if } \text{instance}(X, \text{move}), \text{actor}(X) = A, \text{dest}(X) = \text{loc\_in}(A). \)

\( \text{impossible} \text{ occurs}(X) \text{ if } \text{instance}(X, \text{move}), \text{actor}(X) = A, \text{loc\_in}(A) = O, \text{dest}(X) = D, \neg\text{connected}(O, D). \)

\( \text{impossible} \text{ occurs}(X) \text{ if } \text{instance}(X, \text{move}), \text{actor}(X) = A, \text{is\_held}(A). \)

\( \text{impossible} \text{ occurs}(X) \text{ if } \text{instance}(X, \text{carry}), \text{actor}(X) = A, \text{carried\_object}(X) = C, \neg\text{holding}(A, C). \)

For readability, we selected the same names for the theory and its module. This theory will be used in \[\text{Appendix C}\] for the purpose of comparing \( \text{ALM} \) and \( \text{MAD} \).

Finally, the semantics of a system description with a theory \( T \) consisting of multiple modules is given by the collection of models of the \( \text{BAT} \) defined by \( f(T) \) and the collection of interpretations defined by the system’s structure.

This concludes our introduction to the syntax and semantics of \( \text{ALM} \).

4 Methodology of Language Use

In this section we further illustrate the methodology of using \( \text{ALM} \) for knowledge representation and for solving various computational tasks.

4.1 Representing Knowledge in \( \text{ALM} \)

We exemplify the methodology of representing knowledge in \( \text{ALM} \) by considering a benchmark commonsense example from the field of reasoning about action and change — the Monkey and Banana Problem [McCarthy 1963] [McCarthy 1968]. (Another, more realistic, example of the use of \( \text{ALM} \) can be found in [Appendix B].)
Problem 1 (Monkey and Banana)
A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey’s reach. In the room there is also a box. The ceiling is just the right height so that a monkey standing on the box under the bananas can reach the bananas. The monkey can move around, carry other things around, climb on the box, and grasp the bananas. What is the best sequence of actions for the monkey to get the bananas?

In accordance with the basic methodology of declarative programming, we will first represent knowledge about the problem domain and then reduce the problem’s solution to reasoning with this knowledge. Based on our current experience, we recommend to divide the process of representation into the following steps:

**Methodology of Creating Modular Representations in ALM:**

- Build a hierarchy of actions pertinent to the domain.
- Starting from the top of the hierarchy gradually build and test modules capturing properties of its actions. If necessary, add general non-action modules (e.g. a module defining a sequence of actions). Whenever feasible, use existing library modules.
- Build a module main containing specific information needed for the problem solution.
- Populate the hierarchy with the domain’s objects.

Here are a few comments about the second step listed above: When deciding how many actions to describe in one module, consider balancing the size of the module with the depth of the (part of the) hierarchy that it captures; also consider the resulting depth of the module dependency hierarchy. For instance, an action and its opposite are normally included in the same module. So are actions that usually occur together and share common fluents and sorts. To facilitate the discovery of relevant library modules, we assume that a dictionary indexed by action classes will be available to knowledge engineers. Action classes will be associated with the library modules in which they are described. The signature and axioms of library modules will be viewable by the knowledge engineer.

Let us illustrate the methodology by solving the Monkey and Banana problem. The story is clearly about an agent moving around, and grasping and carrying things between various points. The hierarchy of actions pertinent to the story is illustrated in Figure 5.

Note that, unlike other actions, action release does not explicitly appear in the story. However, it is often advisable to consider actions together with their opposites, so our hierarchy contains release together with grasp.

To gradually build a theory monkey and banana containing the knowledge needed to solve the Monkey and Banana problem, we start with selecting a root of the action hierarchy – in our case action move. The inheritance hierarchy pertinent to move appears in Figure 2. We already discussed the module moving describing the properties of move. The theory consisting of this module can be tested on a number
of specific domains using ASP-based methods discussed in the next section. Next we select three actions carry, grasp, and release understood as move while holding, take and hold, and stop holding respectively. Since these actions share a fluent holding and sorts things and agents, and since a things-carrying agent usually also executes actions grasp and release, knowledge about these actions can be put in the same module. To do that we extend the inheritance hierarchy by a subclass carriables of things and expand module carrying_things from section 3.2 by information about another two actions. Sort declarations of carrying_things from section 3.2 will now also include

\[
\text{grasp :: actions}
\]
\[
\text{attributes}
\]
\[
\text{grasper : agents}
\]
\[
\text{grasped_thing : things}
\]

and

\[
\text{release :: actions}
\]
\[
\text{attributes}
\]
\[
\text{releaser : agents}
\]
\[
\text{released_thing : things}
\]

The section function declarations of the new module will contain the additional function can_reach needed as a precondition for the executability of grasp. The function will be defined in terms of locations of things.

\[
\text{defined}
\]
\[
\text{can_reach : agents x things -> booleans}
\]

The set of axioms will be expanded as follows. The first two axioms below describe the direct effects of our new actions: action grasp results in the grasper holding the thing he grasped; this is no longer true after the thing is released.

\[
\text{occurs}(A) \text{ causes } holding(X, Y) \text{ if } \begin{cases}
\text{instance}(A, \text{grasp}), \\
\text{grasper}(A) = X, \\
\text{grasped_thing}(A) = Y.
\end{cases}
\]

\[
\text{occurs}(A) \text{ causes } \neg holding(X, Y) \text{ if } \begin{cases}
\text{instance}(A, \text{release}), \\
\text{releaser}(A) = X, \\
\text{released_thing}(A) = Y.
\end{cases}
\]

\[7\text{ For simplicity we assume that an agent can only hold one thing at a time. A more general module may allow to grasp a collection of things up to a certain capacity.}\]
The constraint
\[
\neg\text{holding}(X, Y_2) \iff \text{holding}(X, Y_1), Y_1 \neq Y_2
\]
ensures that only one thing can be held at a time (and hence to grasp a thing an agent must have his hands free).

This is followed by the executability conditions: one cannot grasp a thing he is already holding or a thing that is out of his reach; one cannot release a thing unless he is holding it.

\[
\text{impossible occurs}(A) \text{ if } \begin{align*}
\text{instance}(A, \text{grasp}), \\
\text{grasper}(A) = X, \\
\text{grasped}_\text{thing}(A) = Y, \\
\text{holding}(X, Y).
\end{align*}
\]

\[
\text{impossible occurs}(A) \text{ if } \begin{align*}
\text{instance}(A, \text{grasp}), \\
\text{grasper}(A) = X, \\
\text{grasped}_\text{thing}(A) = Y, \\
\neg\text{can}_\text{reach}(X, Y).
\end{align*}
\]

\[
\text{impossible occurs}(A) \text{ if } \begin{align*}
\text{instance}(A, \text{release}), \\
\text{releaser}(A) = X, \\
\text{released}_\text{thing}(A) = Y, \\
\neg\text{holding}(X, Y).
\end{align*}
\]

We also need a simple definition of \(\text{can}_\text{reach}\) – an agent can always reach an object he shares a location with.

\[
\text{can}_\text{reach}(M, O) \iff \text{loc}_\text{in}(M) = \text{loc}_\text{in}(O).
\]

This definition will later be expanded to describe the specific geometry of our domain.

This completes our construction of the new module \textit{carrying things}.

After testing the theory consisting of \textit{moving} and \textit{carrying things} we proceed to constructing a new module, \textit{climbing}, which axiomatizes action \textit{climb} understood as \textit{moving from the bottom of a thing to its top}. We assume that one can climb only on tops of a special type of things called \textit{elevations}, which will be added to our hierarchy as a subset of \textit{things}. The corresponding declarations look as follows:

\begin{verbatim}
module climbing
  depends on  moving

sort declarations
  elevations :: things
  climb :: move

attributes
  elevation : elevations
\end{verbatim}

Now we introduce notation for \textit{points associated with the tops of elevations}. The points are represented by \textit{object constants} of the form \(\text{top}(E)\) where \(E\) is an elevation. In \(\text{ALM}\) this is expressed by the following:

\begin{verbatim}
object constants
  \text{top}(\text{elevations}) : points
\end{verbatim}

32  Daniela Incelean and Michael Gelfond
(Notice that \textit{top} here is not a function symbol; if \textit{e} is an elevation, then \textit{top(e)} is simply a point.)

The module contains axioms saying that \textit{top(E)} is the destination of climbing an elevation \textit{E}:

\[
dest(A) = \text{top}(E) \text{ if } \text{elevation}(A) = E.
\]

and that a thing cannot be located on its own top:

\[
\text{false if } \text{loc.in}(E) = \text{top}(E).
\]

The last axiom prohibits an attempt by an agent to climb an elevation from a distance:

\[
\text{impossible occurs}(X) \text{ if } \begin{array}{l}
\text{instance}(X, \text{climb}), \\
\text{actor}(X) = A, \\
\text{elevation}(X) = O, \\
\text{loc.in}(O) \neq \text{loc.in}(A).
\end{array}
\]

After testing the existing modules we concentrate on the specific information needed for the problem solution. It will be presented in a module called \textit{main}.

\textbf{module main}

\textbf{depends on carrying\_things, climbing}

The main goal of the module is to define when the monkey can reach the banana. We start by dividing our sort \textit{points} into three parts: \textit{floor\_points}, \textit{ceiling\_points}, and \textit{movable\_points}:

\textbf{sort declarations}

\textit{floor\_points, ceiling\_points, movable\_points :: points}

where the latter correspond to tops of movable objects. We will see the use of these sorts a little later. Now we move to function declarations. The story is about three particular entities: the monkey, the banana, and the box. They will be defined as constants of our module.

\textbf{object constants}

\texttt{monkey : agents}

\texttt{box : carriables, elevations}

\texttt{banana : carriables}

We will also need a function \textit{under}, such that \textit{under}(P, T) is true when point \textit{P} is located under the thing \textit{T}. Note that, if we consider this function to be defined for arbitrary points, it will be dynamic – \textit{under(top(box), banana)} can be true in one state and false in another. This will force us to declare this function as a fluent, causing an unnecessary complication. Instead we define \textit{under} for floor points only, which is sufficient for our purpose and is substantially simpler.

\textbf{function declarations}

\textbf{statics}

\texttt{basic \textit{under} : floor\_points × things → booleans}

To define our function \textit{can\_reach} we need the following axiom:
Daniela Inclezan and Michael Gelfond

axioms
can_reach(monkey, banana) if 
loc_in(box) = P,
under(P, banana),
loc_in(monkey) = top(box).

Finally, we need the following axioms for the basic fluent connected:
connected(top(box), P) if 
loc_in(box) = P,
instance(P, floor_points).
¬connected(top(box), P) if 
loc_in(box) \neq P,
instance(P, floor_points).
connected(P_1, P_2) if 
instance(P_1, floor_points),
instance(P_2, floor_points).
¬connected(P_1, P_2) if 
instance(P_1, ceiling_points),
instance(P_2, points),
P_1 \neq P_2.

This completes the construction of module main as well as theory monkey_and_banana that we will use to solve the Monkey and Banana problem. It is easy to see that the theory is semantically coherent, as it satisfies the conditions in Definition 11.

Figure 6 and 7 represent the sort hierarchy and module hierarchy of this theory, respectively.

```
Fig. 6. Sort Hierarchy for the Monkey and Banana Problem
```

To complete the description of our domain we introduce the structure containing three points located on the floor of the room and one point located on the ceiling, as well as movable points and particular actions mentioned in the story:

structure monkey_and_banana

instances
under_banana, initial_monkey, initial_box in floor_points
initial_banana in ceiling_points
top(box) in movable_points
move\( (P) \) in move where instance\( (P, points) \)
   actor = monkey
   dest = P

carry\( (\text{box}, P) \) in carry where instance\( (P, \text{floor\_points}) \)
   actor = monkey
   carried\_object = box
   dest = P

grasp\( (C) \) in grasp where instance\( (C, \text{carriables}) \)
   grasper = monkey
   grasped\_thing = C

release\( (C) \) in release where instance\( (C, \text{carriables}) \)
   releaser = monkey
   released\_thing = C

climb\( (\text{box}) \) in climb
   actor = monkey
   elevation = box

values of statics

under\( (\text{under\_banana, banana}) \).

symmetric\_connectivity.
~transitive\_connectivity.

The structure specifies that the relation connected is symmetric, but not transitive.
The latter prevents the monkey from moving from its initial location directly on top of the box.

The theory and structure described above can be combined into a system description monkey\_and\_banana as follows:

system description monkey\_and\_banana\_problem
  theory monkey\_and\_banana
    import motion from commonsense_library
  module main
    (module body)
structure monkey_and_banana
  (structure body)

Note that the import statement above is a directive to import all of the modules of the library theory \textit{motion} into the theory \textit{monkey_and_banana}.

The system describes a unique hierarchy and a unique transition diagram, $\tau$. Note that the hierarchy contains properly typed constants \textit{monkey}, \textit{box}, and \textit{banana} declared in our module \textit{main}; and that some of our functions, e.g. \textit{under}, are partial.

It is not difficult to check that there is a path in $\tau$ that starts with the initial state of our problem and is generated by actions \textit{move(initial\_box)}, \textit{grasp(box)}, \textit{carry(box, \textit{under}\_\textit{banana})}, \textit{release(box)}, \textit{climb(box)}, \textit{grasp(banana)}. The final state of this path will contain a fluent \textit{holding(monkey, banana)}. In the next section we discuss how ASP based reasoning can be used to automatically find such sequences.

4.2 \textit{ALM}'s Use in Solving Computational Tasks

A system description of \textit{ALM} describes a collection of transition diagrams that specifies some dynamic system. System descriptions can be used to solve computational tasks such as temporal projection or planning, using a methodology similar to that developed for non-modular action languages like \textit{AL} (see, for instance, (Gelfond and Kahl 2014)).

4.2.1 Temporal Projection

Normally, system descriptions of \textit{ALM} are used in conjunction with the description of the system’s \textit{recorded history} — a collection of facts about the values of fluents and the occurrences of actions at different time steps in a trajectory. (Since we are only dealing with discrete systems such steps are represented by non-negative integers). Together, the system description and the history define the collection of possible trajectories of the system up to the current step. In our methodology of solving temporal projection tasks, possible trajectories are obtained by computing the answer sets of a logic program. To formally describe this methodology, we need the following definitions.

\textbf{Definition 12} \textit{(History – adapted from Balducci and Gelfond 2003a)}

By the \textit{recorded history} $\Gamma_n$ of a system description $\mathcal{D}$ up to time step $n$ we mean a collection of observations, i.e., facts of the form:

1. $\text{observed}(f(\overline{t}), v, i)$ – fluent $f(\overline{t})$ was observed to have value $v$ at time step $i$, where $0 \leq i \leq n$.
2. $\text{happened}(a, i)$ – action $a$ was observed to happen at time step $i$, where $0 \leq i < n$.

(There are two small differences between this and the definition of a history by Balducini and Gelfond (2003a): the latter only allows boolean fluents and observations that have the form \textit{observed}(l, i) where $l$ is a fluent or its negation. Similarly for the next definitions in this subsection.)
We say that the initial situation of \( \Gamma_n \) is complete if, for every user-defined basic fluent \( f \) and any sequence of ground terms \( \overline{t} \) such that \( \text{observed}(\text{dom}_f(\overline{t}), \text{true}, 0) \in \Gamma_n \), \( \Gamma_n \) also contains a fact of the form \( \text{observed}(f(\overline{t}), v, 0) \).

Example 9 (History)
A possible recorded history for the system description \texttt{monkey_and_banana_problem} in Section 4.1 may look as follows:

\[
\Gamma_1 = \text{def} \{ \text{observed}(\text{loc}_\text{in}(\text{monkey}), \text{initial_monkey}, 0), \\
\text{observed}(\text{loc}_\text{in}(\text{box}), \text{initial_box}, 0), \\
\text{happened}(\text{move}(\text{initial_box}), 0) \}
\]

which says that, initially, the monkey was at point \text{initial_monkey} and the box was at \text{initial_box}; the monkey went to the initial location of the box.

The semantics of a history \( \Gamma_n \) is given by the following definition:

Definition 13 (Model of a History – adapted from (Balduccini and Gelfond 2003a))

Let \( \Gamma_n \) be a history of a system description \( \mathcal{D} \) up to time step \( n \).

(a) A trajectory \( \langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle \) is a model of \( \Gamma_n \) if:

1. \( a_i = \{ a : \text{happened}(a, i) \in \Gamma_n \} \), for every \( 0 \leq i < n \).
2. if \( \text{observed}(f(\overline{t}), v, i) \in \Gamma_n \) then \( f(\overline{t}) = v \in \sigma_i \), for every \( 0 \leq i \leq n \).

(b) \( \Gamma_n \) is consistent if it has a model.

(c) An atom \( f(\overline{t}) = v \) holds in a model \( M \) of \( \Gamma_n \) at time \( 0 \leq i \leq n \) if \( f(\overline{t}) = v \in \sigma_i \);

A literal \( f(\overline{t}) \neq v \) holds in a model \( M \) of \( \Gamma_n \) at time \( 0 \leq i \leq n \) if \( \text{dom}_f(\overline{t}) = \text{true} \in \sigma_i \) and \( f(\overline{t}) = v \notin \sigma_i \);

\( \Gamma_n \) entails a literal \( l \) at time step \( 0 \leq i \leq n \) if, for every model \( M \) of \( \Gamma_n \), \( l \) holds in \( M \).

Example 10 (Model of a History)
History \( \Gamma_1 \) from Example 9 is consistent. Its model is the trajectory:

\[
M = \langle \{ \text{loc}_\text{in}(\text{monkey}) = \text{initial_monkey}, \text{loc}_\text{in}(\text{box}) = \text{initial_box}, \ldots \}, \\
\text{move}(\text{initial_box}), \\
\{ \text{loc}_\text{in}(\text{monkey}) = \text{initial_box}, \text{loc}_\text{in}(\text{box}) = \text{initial_box}, \ldots \} \rangle.
\]

(We do not show the values of \texttt{connected} since they are unchanged by our actions). \( \Gamma_1 \) entails, for example, \texttt{loc}_\text{in}(\texttt{monkey}) = \texttt{initial_box} at time step 1.

Note that a consistent history may have more than one model if non-deterministic actions are involved or the initial situation is not complete.

Next, we define some useful vocabulary.
Definition 14 (Set of Literals Defining a Sequence – adapted from [Balduccini and Gelfond 2003a])

Let $\Gamma_n$ be a history of $D$ and $A$ be a set of literals over signature $\Sigma$. We say that $A$ defines the sequence $\langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ if:

(a) $\sigma_i = \{ f(t_0, \ldots, t_n) = t : f(t_0, \ldots, t_n) = t \in A \text{ and } f \text{ is a static or attribute} \} \cup \{ f(t_0, \ldots, t_n) = t : f(t_0, \ldots, t_n, i) = t \in A \text{ and } f \text{ is a fluent} \}$ for any $0 \leq i \leq n$, and

(b) $a_k = \{ a : \text{occurs}(a, k) \in A \}$ for any $0 \leq k < n$.

Definition 15 (Program $\Omega_{tp}$ – adapted from [Balduccini and Gelfond 2003a])

If $\Gamma_n$ is a history of system description $D$ up to time step $n$, then by $\Omega_{tp}$ we denote the ASP $\{ f \}$ program constructed as follows:

1. For every action $a$ such that $\text{happened}(a, i) \in \Gamma_n$, $\Omega_{tp}$ contains:

   $\text{occurs}(a, i) \leftarrow \text{happened}(a, i)$.

2. For every expression $\text{observed}(f(\overline{t}), v, 0) \in \Gamma_n$, $\Omega_{tp}$ contains:

   $f(\overline{t}, 0) = v \leftarrow \text{observed}(f(\overline{t}), v, 0)$.

3. For every expression $\text{observed}(f(\overline{t}), v, i) \in \Gamma_n$, $i > 0$, $\Omega_{tp}$ contains the reality check axiom:

   $\leftarrow \text{observed}(f(\overline{t}), v, i),
   \text{dom}(\overline{t}, i),$
   $f(\overline{t}, i) \neq v$.

Our methodology of finding trajectories by computing answer sets of a logic program is designed for system descriptions that match the intuition that defined functions are only shorthands, and their values are fully determined by those of basic statics and fluents. We call such system descriptions well-founded and define them formally as follows.

Definition 16 (Well-founded System Description – adapted from [Gelfond and Inclezan 2013])

Let $D$ be a system description whose theory encodes the $\mathcal{BAT}$ theory $T$, and whose structure defines a collection $S$ of models of $T$. $D$ is well-founded if, for every model $M$ in $S$, and every interpretation $I$ in $M$, the program $S_I$ (defined as in Section 2.3) has at most one answer set.

The system description monkey_and_banana_problem from Section 4.1 is well-founded. An example of a system description that is not well-founded is $n_w_f$ shown below and adapted from [Gelfond and Inclezan 2013]. The two defined fluents of $n_w_f$ are not defined in terms of basic statics or fluents but rather in terms of one another by mutually recursive axioms.
system description \textit{n.w.f}
theory \textit{n.w.f}
module \textit{main}
  sort declarations
  \textit{c :: universe}
fluent declarations
defined
  \( f : c \rightarrow \text{booleans} \)
  \( g : c \rightarrow \text{booleans} \)
axioms
  \( f(X) \text{ if } \neg g(X). \)
  \( g(X) \text{ if } \neg f(X). \)
structure \textit{n.w.f}
instances
\textit{x in c}

In the case of the non-modular action language \textit{AL}, there is a known syntactic condition that guarantees that a system description is well-founded \cite{GelfondInclezan2013}. This condition can be easily expanded to \textit{ALM} due to close connections between \textit{ALM} and \textit{AL}.

Trajectories of a dynamic system specified by a well-founded system description are computed using a logic program \( \Pi \) that consists of the ASP \{\textit{f}\} encoding of the system description, the system’s recorded history, and the program \( \Omega_{tp} \) connecting the recorded history with the system description.

To simplify the presentation, in what follows we limit ourselves to well-founded system descriptions that describe domains in which there is complete information about the sort memberships of objects of the domain.\footnote{This is not a serious restriction; it can be easily lifted by adding to the ASP encoding of the \textit{ALM} system description rules of the type \( \text{is.a}(x, c) \text{ or } \neg \text{is.a}(x, c) \) for every object \( x \) and every source node \( c \) in the hierarchy of sorts.} Let us consider system description \( D \) that meets this requirement, and let \( \mathcal{M} \) be a model of \( D \)’s theory. Then, the program \( \Pi_{tp} \) obtained from the theory of \( D \) and some interpretation \( I \) of \( \mathcal{M} \) as described in section \ref{sec:encoding} will be used as the ASP\{\textit{f}\} encoding of \( D \).

\textit{Definition 17 (Program \( \Pi_{tp}(D) \))}
If \( \Gamma_n \) is a history of \( D \) up to step \( n \), then \( \Pi_{tp}(D) \) is the logic program defined as
\[
\Pi_{tp}(D) \equiv_{def} P_I \cup \Gamma_n \cup \Omega_{tp}
\]
such that the sort \textit{step} in the signature of \( \Pi_{tp}(D) \) ranges over the set \( \{0, \ldots, n\} \).

\textit{Proposition 1}
If \( \Gamma_n \) is a consistent history of \( D \) such that the initial situation of \( \Gamma_n \) is complete, then \( \mathcal{M} \) is a model of \( \Gamma_n \) iff \( \mathcal{M} \) is defined by some answer set of program \( \Pi_{tp}(D) \).
This proposition can be proven using techniques similar to the ones employed in Lemma 5 in [Balduccini and Gelfond 2003a].

We used the above methodology of solving temporal projection tasks to create a question answering system in the context of the Digital Aristotle project [Inclezan and Gelfond 2011]. Our system was capable of answering complex end-of-the-chapter questions on cell division, extracted from a well-known biology textbook.

4.2.2 Planning

In planning problems, in addition to the history of the dynamic system up to the current time point, information about the goal to be achieved is also provided. Given a system description of $ALM$ whose theory describes a basic action theory $T$, a goal is a collection $G$ of ground user-defined fluent literals over the signature of $T$. For instance, for the Monkey and Banana problem in Section 4.1, the goal is $G_{mb} = \{\text{holding}(\text{monkey}, \text{banana})\}$. Goals can be encoded as logic programming rules, as described in the following definition:

**Definition 18 (Goal Encoding)**

Given a goal $G$, we call encoding of $G$, denoted by $lp(G)$ the rule

$$
goal(I) \leftarrow \text{body}$$

where $\text{body}$ is defined as follows:

$$\text{body} = \text{def} \{f(t, I) = v \ : \ f(t) = v \in G\} \cup \{f(t, I) \neq v \ : \ f(t) \neq v \in G\}.$$ 

In order to solve planning problems, a slightly different logic programming module will be needed than for solving temporal projection tasks. This module is defined in CR-Prolog [Balduccini and Gelfond 2003b], an extension of ASP designed to handle, among other things, rare events. In addition to regular ASP rules, programs in CR-Prolog may contain consistency restoring rules that have the following syntax:

$$h_1 \lor \ldots \lor h_k \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n.$$ 

Informally, this statement says that an intelligent agent who believes $l_1, \ldots, l_m$ and has no reason to believe $l_{m+1}, \ldots, l_n$ may believe one of $h_i$’s, $1 \leq i \leq k$, but only if no consistent set of beliefs can be formed otherwise. For the formal semantics of CR-Prolog, we refer the reader to [Balduccini and Gelfond 2003b]. An extension of ASP{$f$} by consistency restoring rules is defined in [Balduccini and Gelfond 2012]. Solvers for CR-Prolog are described in [Balduccini 2007] and [Balai et al. 2012].

**Definition 19 (Planning Module [Balduccini 2004; Gelfond and Kahl 2014])**

---

Given a goal $G$, the planning module $\Omega_{pl}$ extends module $\Omega_{tp}$ from Section 4.2.1 by the following rules:

\[
\begin{align*}
\text{success} & \leftarrow \text{goal}(I), I \leq n \\
& \leftarrow \text{not success} \\
r_1(A,I) : \quad \text{occurs}(A,I) & \leftarrow \text{instance}(A, \text{actions}) \\
\text{smtg\_happened}(I) & \leftarrow \text{occurs}(A,I) \\
& \leftarrow \text{not smtg\_happened}(I), \\
& \quad \text{smtg\_happened}(I + 1)
\end{align*}
\]

$\Omega_{pl}$ computes minimal plans of maximum length $n$ by the use of the consistency restoring rule $r_1$ and the two regular rules that follow it.

The actual program for computing plans is constructed similarly as before.

**Definition 20 (Program $\Pi_{pl}(D)$)**

If $\Gamma_n$ is a history of $D$ up to step $n$ and $G$ is a goal over $D$, then $\Pi_{pl}(D, G)$ is the logic program defined as

\[\Pi_{pl}(D, G) = \text{def} P_\Pi \cup \Gamma_n \cup \Omega_{pl} \cup lp(G)\]

such that the sort $\text{step}$ in the signature of $\Pi_{pl}(D, G)$ ranges over the set $\{0, \ldots, n\}$.

The following proposition specifies how answer sets of the logic program defined above can be mapped into plans for achieving given goals.

**Proposition 2**

If $\Gamma_n$ is a consistent history of $D$ such that the initial situation of $\Gamma_n$ is complete and $G$ is a goal over $D$, then the collection of atoms of the form $\text{occurs}(a, i)$ from an answer set of $\Pi_{pl}(D, G)$ defines a minimal plan for achieving goal $G$, and every such plan is represented by the $\text{occurs}$ atoms of some answer set of $\Pi_{pl}(D, G)$.

**Example 11 (Planning in the Monkey and Banana Problem)**

If we consider the Monkey and Banana problem with the initial situation

\[\Gamma_{mb} = \{ \text{observed(loc\_in(monkey), initial\_monkey, 0)}, \text{observed(loc\_in(box), initial\_box, 0)} \}

and the goal

\[G_{mb} = \{ \text{holding(monkey, banana)} \} \]

defined earlier, then an answer set of program $\Pi_{pl}(\text{monkey\_and\_banana\_problem}, G_{mb})$ will contain the following $\text{occurs}$ atoms:

\[
\{ \text{occurs(move(initial\_box), 0)}, \quad \text{occurs(grasp(box), 1)}, \\
\text{occurs(carry(box, under\_banana), 2)}, \quad \text{occurs(release(box), 3)}, \\
\text{occurs(climb(box), 4)}, \quad \text{occurs(grasp(banana), 5)} \}
\]

defining a minimal plan $\langle \text{move(initial\_box), grasp(box), carry(box, under\_banana), release(box), climb(box), grasp(banana)} \rangle$ resulting in the monkey holding the banana at time step 6. The program will also find the second minimal plan in which $\text{carry(box, under\_banana)}$ at step 2 is replaced by $\text{move(under\_banana)}$. Since the
first action is more specific than the second one the first plan seems to be preferable. This can easily be expressed by a slightly modified planning module allowing only most specific actions.

5 Related Work

Many ideas of \(\textit{ALM}\), such as the notions of action language, module, sort hierarchy, attribute defined as a partial function, etc., are well-known from the literature on programming languages and knowledge representation. Some of the basic references to these notions were given in the text. In this section we briefly comment on the relationship between \(\textit{ALM}\) and the previously existing modular action languages \(\textit{MAD}\) [Lifschitz and Ren 2006], [Erdogan and Lifschitz 2006], [Desai and Singh 2007], TAL-C [Gustafsson and Kvarnstrom 2004], and the earlier version of \(\textit{ALM}\) [Gelfond and Inclezan 2009].

We start with summarizing the differences between the two versions of \(\textit{ALM}\). There are a number of changes in the syntax of the language. For instance, theories of the new version of \(\textit{ALM}\) may contain non-boolean fluents\(^\text{10}\) and constants that substantially simplify \(\textit{ALM}\)'s use for knowledge representation. Axioms of a theory, which in the old version were included in the theory’s declarations, are now put in a separate section of the theory. This removed the problem of deciding which fluent or action declaration should contain an axiom, and improved the readability of the language. There are also substantial improvements in the syntax of axioms, etc.

Another collection of changes is related to the semantics of the language. First, the new semantics, based on the notions of basic action theory and its models, clarified and generalized the old definition and allowed the introduction of the entailment relation. Second, the semantics is now defined for structures with possibly under-specified membership relations of its objects in the sort hierarchy, which simplifies reasoning with incomplete information. Third, the semantics was initially given in terms of action language \(\textit{AL}\) [Turner 1997], [Baral and Gelfond 2000], where the \(\textit{AL}\) semantics is defined by a translation into ASP; now, we give the semantics of our language directly in ASP – in fact, in an extension of ASP with non-Herbrand functions, ASP\(\{f\}\) [Balduccini 2013]. We believe that decoupling \(\textit{ALM}\) from \(\textit{AL}\) will allow us to combine \(\textit{ALM}\) with action languages that correspond to other intuitions.

Another modular language is TAL-C [Gustafsson and Kvarnstrom 2004], which allows definitions of classes of objects that are somewhat similar to those in \(\textit{ALM}\). TAL-C, however, seems to have more ambitious goals: the language is used to describe and reason about various dynamic scenarios, whereas in \(\textit{ALM}\) the description of a scenario and that of reasoning tasks are not viewed as part of the language. The more rigid structure of \(\textit{ALM}\) supports the separation of concerns design principle and makes it easier to give a formal semantics of the language.

\(^{10}\) In the field of logic programming, an early discussion on the introduction of functions appears in [Hanus 1994].
These differences led to vastly distinct knowledge representation styles reflected in these languages.

There are smaller, but still very substantial, differences between ALM and MAD. The two languages are based on non-modular action languages with substantially different semantics and underlying assumptions, use very different constructs for creating modules and for defining actions as special cases, etc. A more detailed comparison between the two approaches can be found in the Appendix C.

6 Conclusions and Future Work

In this paper, we have presented a methodology of representing and reasoning about dynamic systems. A knowledge engineer following this methodology starts with finding a proper generalization of a particular dynamic system $D$, finds the sorts of objects pertinent to this generalization, organizes these sorts into an inheritance hierarchy and uses causal laws, definitions, and executability conditions to specify relevant properties of the sorts elements. The resulting basic action theory, say $T$, gives the first mathematical model of the system. In the next step of the development, a knowledge engineer refines this model by providing its description in the high level action language ALM. The language has means for precisely representing the signature of $T$ including its sort hierarchy. It is characterized by a modular structure, which improves readability and supports the step-wise development of a knowledge base, reuse of knowledge, and creation of knowledge libraries. ALM’s description of $T$ can be used to specify multiple dynamic systems with different collections of objects and statics. A particular system $D$ can be specified by populating sorts of $T$ by objects of $D$ and defining values of $D$’s statics. This step is also supported by ALM, which clearly separates the definition of sorts of objects of the domain (given in $T$) from the definition of instances of these sorts (given by an ALM structure). This, together with the means for defining objects of the domain as special cases of previously defined ones, facilitates the stepwise development and testing of the knowledge base and improves its elaboration tolerance.

A close relationship between ALM and Answer Set Programming allows the use of ALM system descriptions for non-trivial reasoning problems including temporal projection, planning, and diagnosis. This is done by an automatic translation of an ALM system description into logic programs whose answer sets correspond to solutions of the corresponding problems. The existence of efficient answer set solvers that allow to compute these answer sets substantially increases the practical value of this approach.

The above methodology has been illustrated by two examples: the well-known benchmark Monkey and Banana problem and a more practical problem of formalization of knowledge and answering questions about biological processes such as the cell division (see Appendix B). It is possible (and even likely) that further experience with ALM will suggest some useful extensions of the language but the authors believe that the version presented in this paper will remain relatively stable and provide a good basis for such extensions.
We conclude by briefly outlining a number of questions about $\mathcal{ALM}$ that we believe deserve further investigation:

- Investigating mathematical properties of $\mathcal{ALM}$ and its entailment relation. This includes but is not limited to studying compositional properties of $\mathcal{ALM}$ modules, axiomatizing its entailment relation, and establishing a closer relationship between $\mathcal{ALM}$ and modular logic programming.
- Developing more efficient reasoning algorithms exploiting the modular structure of $\mathcal{ALM}$’s theories and the available information about the sorts of objects in $\mathcal{ALM}$’s system descriptions. Among other things it is worth investigating the possible use of modular logic programming as well as the methods from (Gebser et al. 2011), (Gebser et al. 2011), and (Balai et al. 2012). It may also be interesting to see if the implementation could benefit from hybrid approaches combining description logics with ASP (e.g. (Eiter et al. 2008)) or from typed logic programming (e.g. (Pfenning 1992)).
- Designing and implementing a development environment to facilitate the use of $\mathcal{ALM}$ in applications, the creation and storage of libraries, and the testing and debugging of theories and modules.
- Extending $\mathcal{ALM}$ with the capability of representing knowledge about hybrid domains, i.e., domains that allow both discrete and continuous change. In particular, it may be a good idea to combine $\mathcal{ALM}$ with action language $\mathcal{H}$ (Chintabathina et al. 2005; Chintabathina 2012).
- Developing the core of an $\mathcal{ALM}$ library of commonsense knowledge. (In particular we would like to create an $\mathcal{ALM}$ library module containing a theory of intentions in the style of (Blount et al. 2014).) This work would allow us to extend our study on the capabilities of our language, while simultaneously providing a tool for members of our community to use when building their reasoning systems.

Acknowledgments

We are grateful to Evgenii Balai, Justin Blount, Vinay Chaudhri, Vladimir Lifschitz, Yana Todorova, and the anonymous reviewers for useful comments and discussions. This work was partially supported by NSF grant IIS-1018031.
Appendix A  Grammar of ALM

\[
\langle boolean \rangle :: \text{true} \mid \text{false}
\]
\[
\langle \text{non\_zero\_digit} \rangle ::= 1 \mid \cdots \mid 9
\]
\[
\langle \text{digit} \rangle ::= 0 \mid \langle \text{non\_zero\_digit} \rangle
\]
\[
\langle \text{lowercase\_letter} \rangle ::= \text{a} \mid \cdots \mid \text{z}
\]
\[
\langle \text{uppercase\_letter} \rangle ::= \text{A} \mid \cdots \mid \text{Z}
\]
\[
\langle \text{letter} \rangle ::= \langle \text{lowercase\_letter} \rangle \mid \langle \text{uppercase\_letter} \rangle
\]
\[
\langle \text{identifier} \rangle ::= \langle \text{lowercase\_letter} \rangle \mid \langle \text{identifier}\rangle\langle \text{letter} \rangle \mid \langle \text{identifier}\rangle\langle \text{digit} \rangle
\]
\[
\langle \text{variable} \rangle ::= \langle \text{uppercase\_letter} \rangle \mid \langle \text{variable}\rangle\langle \text{letter} \rangle \mid \langle \text{variable}\rangle\langle \text{digit} \rangle
\]
\[
\langle \text{positive\_integer} \rangle ::= \langle \text{non\_zero\_digit} \rangle \mid \langle \text{positive\_integer}\rangle\langle \text{digit} \rangle
\]
\[
\langle \text{integer} \rangle ::= 0 \mid \langle \text{positive\_integer} \rangle \mid - \langle \text{positive\_integer} \rangle
\]
\[
\langle \text{arithmetic\_op} \rangle ::= + \mid - \mid * \mid / \mid \text{mod}
\]
\[
\langle \text{comparison\_rel} \rangle ::= > \mid \geq \mid < \mid \leq
\]
\[
\langle \text{arithmetic\_rel} \rangle ::= (\text{eq}) \mid (\text{neq}) \mid \langle \text{comparison\_rel} \rangle
\]
\[
\langle \text{basic\_arithmetic\_term} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{identifier} \rangle \mid \langle \text{integer} \rangle
\]
\[
\langle \text{term} \rangle ::= \langle \text{basic\_term} \rangle \mid \langle \text{boolean} \rangle
\]
\[
\langle \text{function\_term} \rangle ::= \langle \text{identifier} \rangle\langle \text{function\_args} \rangle
\]
\[
\langle \text{function\_args} \rangle ::= (\langle \text{term} \rangle \langle \text{remainder\_function\_args} \rangle)
\]
\[
\langle \text{remainder\_function\_args} \rangle ::= \epsilon \mid , \langle \text{term} \rangle \langle \text{remainder\_function\_args} \rangle
\]
\[
\langle \text{arithmetic\_term} \rangle ::= \langle \text{basic\_arithmetic\_term} \rangle \langle \text{arithmetic\_op} \rangle \langle \text{basic\_arithmetic\_term} \rangle
\]
\[
\langle \text{term} \rangle ::= (\langle \text{basic\_term} \rangle) \mid (\langle \text{arithmetic\_term} \rangle)
\]
\[
\langle \text{positive\_function\_literal} \rangle ::= \langle \text{function\_term} \rangle \langle \text{function\_term} \rangle (\text{eq}) \langle \text{term} \rangle
\]
\[
\langle \text{function\_literal} \rangle ::= \langle \text{positive\_function\_literal} \rangle \mid \langle \text{positive\_function\_literal} \rangle \langle \text{function\_term} \rangle \langle \text{term} \rangle
\]
\[
\langle \text{function\_term} \rangle ::= (\langle \text{term} \rangle \langle \text{remainder\_function\_args} \rangle)
\]
\[
\langle \text{function\_args} \rangle ::= \langle \text{arithmetic\_term} \rangle \langle \text{arithmetic\_rel} \rangle (\langle \text{arithmetic\_term} \rangle)
\]
\[
\langle \text{var\_id} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{identifier} \rangle
\]
\[
\langle \text{body} \rangle ::= \epsilon \mid , \langle \text{literal} \rangle \langle \text{body} \rangle
\]
\[
\langle \text{dynamic\_causal\_law} \rangle ::= \langle \text{occurs} \rangle \langle \text{var\_id} \rangle \langle \text{causes} \rangle \langle \text{positive\_function\_literal} \rangle \text{if} \langle \text{instance} \rangle \langle \text{var\_id} \rangle \langle \text{var\_id} \rangle \langle \text{body} \rangle.
\]
\[
\langle \text{state\_constraint} \rangle ::= \langle \text{sc\_head} \rangle \text{if} \langle \text{body} \rangle.
\]
\[
\langle \text{sc\_head} \rangle ::= \langle \text{positive\_function\_literal} \rangle \text{if} \langle \text{instance} \rangle \langle \text{term} \rangle \text{if} \langle \text{body} \rangle.
\]
\[
\langle \text{definition} \rangle ::= \langle \text{function\_term} \rangle \text{if} \langle \text{body} \rangle.
\]
\[
\langle \text{executability\_condition} \rangle ::= \langle \text{impossible occurs} \rangle \langle \text{var\_id} \rangle \text{if} \langle \text{instance} \rangle \langle \text{var\_id} \rangle \langle \text{var\_id} \rangle \langle \text{extended\_body} \rangle.
\]
\[
\langle \text{extended\_body} \rangle ::= \epsilon \mid , \langle \text{literal} \rangle \langle \text{body} \rangle \mid , \langle \text{occurs} \rangle \langle \text{var\_id} \rangle \langle \text{extended\_body} \rangle \mid 
\]
\[
\langle \text{system\_description} \rangle ::= \langle \text{system\_description} \rangle \langle \text{identifier} \rangle \langle \text{theory} \rangle \langle \text{structure} \rangle
\]
\[
\langle \text{theory} \rangle ::= \langle \text{theory} \rangle \langle \text{identifier} \rangle \langle \text{set\_of\_modules} \rangle \mid \langle \text{import} \rangle \langle \text{from} \rangle \langle \text{identifier} \rangle
\]
\[
\langle \text{set\_of\_modules} \rangle ::= \langle \text{module} \rangle \langle \text{remainder\_modules} \rangle
\]
\[
\langle \text{remainder\_modules} \rangle ::= \epsilon \mid \langle \text{module} \rangle \langle \text{remainder\_modules} \rangle
\]
\[
\langle \text{module} \rangle ::= \langle \text{module} \rangle \langle \text{module\_body} \rangle \mid 
\]
\[
\langle \text{import} \rangle \langle \text{from} \rangle \langle \text{identifier} \rangle \langle \text{identifier} \rangle \langle \text{axioms} \rangle
\]
\[
\langle \text{module\_body} \rangle ::= \langle \text{sort\_declarations} \rangle \langle \text{constant\_declarations} \rangle \langle \text{function\_declarations} \rangle \langle \text{axioms} \rangle
\]
\langle sort\_declarations \rangle ::= \epsilon | sort\_declarations \langle one\_sort\_decl \rangle \langle remainder\_sort\_declarations \rangle
\langle remainder\_sort\_declarations \rangle ::= \epsilon | \langle one\_sort\_decl \rangle \langle remainder\_sort\_declarations \rangle
\langle one\_sort\_decl \rangle ::= \langle identifier \rangle \langle remainder\_sort\_sorts \rangle :: (sort\_name) \langle remainder\_sort\_names \rangle \langle attributes \rangle
\langle remainder\_sorts \rangle ::= \epsilon | (\langle identifier \rangle \langle remainder\_sorts \rangle)
\langle remainder\_sort\_names \rangle ::= \epsilon | (\langle sort\_name \rangle \langle remainder\_sorts \rangle)
\langle sort\_name \rangle ::= (\langle identifier \rangle) | [ [ (\langle integer \rangle .. \langle integer \rangle) ]
\langle attributes \rangle ::= \epsilon | attributes \langle one\_attribute\_decl \rangle \langle remainder\_attribute\_declarations \rangle
\langle one\_attribute\_decl \rangle ::= \langle identifier \rangle : (arguments) \langle identifier \rangle
\langle arguments \rangle ::= \epsilon | (\langle identifier \rangle \langle remainder\_args \rangle) \rightarrow
\langle remainder\_args \rangle ::= \epsilon | × (\langle identifier \rangle \langle remainder\_args \rangle)
\langle remainder\_attribute\_declarations \rangle ::= \epsilon |
\langle one\_attribute\_decl \rangle \langle remainder\_attribute\_declarations \rangle
\langle constant\_declarations \rangle ::= \epsilon | constant\_declarations \langle one\_constant\_decl \rangle \langle remainder\_constant\_declarations \rangle
\langle one\_constant\_decl \rangle ::= \langle identifier \rangle \langle constant\_params \rangle : (identifier)
\langle remainder\_constant\_declarations \rangle ::= \epsilon | \langle one\_constant\_decl \rangle \langle remainder\_constant\_declarations \rangle
\langle constant\_params \rangle ::= (\langle identifier \rangle \langle remainder\_constant\_params \rangle)
\langle remainder\_constant\_params \rangle ::= \epsilon | (\langle identifier \rangle \langle remainder\_constant\_params \rangle)
\langle function\_declarations \rangle ::= \epsilon | function\_declarations \langle static\_declarations \rangle \langle fluent\_declarations \rangle
\langle static\_declarations \rangle ::= \epsilon | statics \langle basic\_function\_declarations \rangle \langle defined\_function\_declarations \rangle
\langle fluent\_declarations \rangle ::= \epsilon | fluents \langle basic\_function\_declarations \rangle \langle defined\_function\_declarations \rangle
\langle basic\_function\_declarations \rangle ::= \epsilon | (\langle function\_decl \rangle \langle remainder\_function\_declarations \rangle)
\langle defined\_function\_declarations \rangle ::= \epsilon | defined \langle one\_function\_decl \rangle \langle remainder\_function\_declarations \rangle
\langle one\_function\_decl \rangle ::= (total\_partial) \langle one\_f\_decl \rangle
\langle total\_partial \rangle ::= \epsilon | total
\langle one\_f\_decl \rangle ::= (\langle identifier \rangle) : (\langle identifier \rangle) \langle remainder\_args \rangle \rightarrow (\langle identifier \rangle)
\langle remainder\_function\_declarations \rangle ::= \epsilon | (\langle one\_function\_decl \rangle \langle remainder\_function\_declarations \rangle)
\langle axioms \rangle ::= \epsilon | axioms \langle one\_axiom \rangle \langle remainder\_axioms \rangle
\langle one\_axiom \rangle ::= (dynamic\_causal\_law) | (state\_constraint) | (definition) | (executability\_condition)
\langle remainder\_axioms \rangle ::= \epsilon | (\langle axiom \rangle) \langle remainder\_axioms \rangle
\langle structure \rangle ::= structure \langle identifier \rangle \langle constant\_defs \rangle \langle instance\_defs \rangle \langle statics\_defs \rangle
\langle constant\_defs \rangle ::= \epsilon | constants \langle one\_constant\_def \rangle \langle remainder\_constant\_defs \rangle
\langle one\_constant\_def \rangle ::= (\langle identifier \rangle) = (\langle value \rangle)
\langle value \rangle ::= (\langle identifier \rangle) | (\langle boolean \rangle) | (\langle integer \rangle)
\langle remainder\_constant\_defs \rangle ::= \epsilon | (\langle one\_constant\_def \rangle \langle remainder\_constant\_defs \rangle)
\langle instance\_defs \rangle ::= \epsilon | instances \langle one\_instance\_def \rangle \langle remainder\_instance\_defs \rangle
\langle one\_instance\_def \rangle ::= (\langle object\_name \rangle) \langle remainder\_object\_names \rangle in
\langle identifier \rangle \langle cond \rangle \langle attribute\_defs \rangle
\langle object\_name \rangle ::= (\langle identifier \rangle) \langle object\_args \rangle
\langle object\_args \rangle ::= \epsilon | ((basic\_term) \langle remainder\_object\_args \rangle)
\langle remainder\_object\_args \rangle ::= \epsilon | (basic\_term) \langle remainder\_object\_args \rangle
\langle remainder\_object\_names \rangle ::= \epsilon | (\langle object\_name \rangle) \langle remainder\_object\_names \rangle
\langle cond \rangle ::= \epsilon | where \langle literal \rangle \langle remainder\_cond \rangle
\langle remainder\_cond \rangle ::= \epsilon | (\langle literal \rangle) \langle remainder\_cond \rangle
Modular Action Language \( \text{ALM} \)

\[
\langle \text{attribute_defs} \rangle :\overset{\epsilon}{\rightarrow} \langle \text{one_attribute_def} \rangle \langle \text{remainder_attribute_defs} \rangle \\
\langle \text{one_attribute_def} \rangle :\overset{\langle \text{identifier} \rangle \langle \text{object_args} \rangle = \langle \text{basic_term} \rangle}{\rightarrow} \\
\langle \text{statics_defs} \rangle :\overset{\epsilon}{\rightarrow} \text{values of statics} \langle \text{one_static_def} \rangle \langle \text{remainder_statics_defs} \rangle \\
\langle \text{one_static_def} \rangle :\overset{\langle \text{functionLiteral} \rangle \text{ if } \langle \text{body} \rangle}{\rightarrow} \\
\langle \text{remainder_statics_defs} \rangle :\overset{\epsilon}{\rightarrow} \langle \text{one_static_def} \rangle \langle \text{remainder_statics_defs} \rangle
\]
Appendix B  ALM and the Digital Aristotle

The reader may have noticed that the ALM examples included in the body of the paper are relatively small, which is understandable given that their purpose was to illustrate the syntax and semantics of our language and the methodology of representing knowledge in ALM. In this section, we show how the reuse of knowledge in ALM can potentially lead to the creation of larger practical systems. We present an application of our language to the task of question answering, in which ALM’s conceptual separation between an abstract theory and its structure played an important role in the reuse of knowledge. The signature of the theory and its structure provided the vocabulary for the logic form translation of facts expressed in natural language while the theory axioms contained the background knowledge needed for producing answers. The theory representing the biological domain remained unchanged and was coupled with various structures corresponding to particular questions and representing the domain at different levels of granularity. In addition to demonstrating the reuse of knowledge in ALM, this application also shows the elaboration tolerance of our language, as only minor changes to the structure had to be made when the domain was viewed in more detail, while the theory stayed the same. In what follows, we present the application in more detail.

After designing our language, we tested and confirmed its adequacy for knowledge representation in the context of a practical question answering application: Project Halo (2002-2013) sponsored by Vulcan Inc.11 The goal of Project Halo was the creation of a Digital Aristotle — “an application containing large volumes of scientific knowledge and capable of applying sophisticated problem-solving methods to answer novel questions” (Gunning et al. 2010). Initially, the Digital Aristotle was only able to reason and answer questions about static domains. It lacked a methodology for answering questions about dynamic domains, as it was not clear how to represent and reason about such domains in the language of the Digital Aristotle. Our task within Project Halo was to create a methodology for answering questions about temporal projection in dynamic domains. We had two objectives. First, we wanted to see if the use of ALM for knowledge representation facilitated the task of encoding extensive amounts of scientific knowledge through its means for the reuse of knowledge. Second, we investigated whether provable correct and efficient logic programming algorithms could be developed to use the resulting ALM knowledge base in answering non-trivial questions.

Our target scientific domain was biology, specifically the biological process of cell division (also called cell cycle). Cell cycle refers to the phases a cell goes through from its “birth” to its division into two daughter cells. Cells consist of a number of parts, which in turn consist of other parts (e.g., eukaryotic cells contain organelles, cytoplasm, and a nucleus; the nucleus contains chromosomes, and the description can continue with more detailed parts). The eukaryotic cell cycle consists of a growth phase (interphase) and a duplication/division phase (mitotic phase), both of which are conventionally described as sequences of sub-phases. Depending on the level

of detail of the description, these sub-phases may be simple events or sequences of other sub-phases (e.g., the mitotic phase is described in more detail as a sequence of two sub-phases: mitosis and cytokinesis; mitosis, in turn, can be seen as a sequence of five sub-phases, etc.). Certain chemicals, if introduced in the cell, can interfere with the ordered succession of events that is the cell cycle.

In order to be useful in answering complex questions, the ALM representation of cell cycle had to capture (1) non-trivial specialized biological knowledge about the structure of the cell at different stages of the cell cycle and (2) the dynamics of naturally evolving process (such as cell cycle), which consist of a series of phases and sub-phases that follow one another in a specific order, unless interrupted. We represented such processes as sequences of actions intended by nature and used a commonsense theory of intentions [Baral and Gelfond 2005] to reason about them.

Our ALM cell cycle knowledge base consisted of two library modules. One of them was a general commonsense module describing sequences, in particular sequences of actions. The other module was a specialized one formalizing the biological phenomenon of cell division. We begin with the presentation of our commonsense module describing sequences, useful in modeling naturally evolving processes such as cell division. The equality component(S, N) = E appearing in the axioms of module sequence is supposed to be read as “the Nth component of sequence S is E”.

The library module sequence is stored in a general library called commonsense_lib.

module sequence

sort declarations

sequences :: universe

attributes

length : positive_natural_numbers

component : [0..length] → universe

action_sequences :: sequences

axioms

false if component(S, N) = E,

instance(S, action_sequences),

¬instance(E, actions),

¬instance(E, action_sequences).

The axiom ensures proper typing for the domain of an attribute component.

Next, we present our formalization of cell cycle, given in a library module called basic_cell_cycle stored in a general cell_cycle_lib library. We started by modeling the eukaryotic cell, consisting of various parts that in turn consist of other parts. Together, they form a “part of” hierarchy, say H_cell, which can be viewed as a tree. Nodes of this hierarchy were captured by a new sort, types_of_parts, while links in the hierarchy were represented by an attribute, is_part_of, defined on elements of the new sort (e.g., is_part_of(X) = Y indicates that Y is the father of X in H_cell). We modeled the transitive closure of is_part_of by introducing a boolean function, part_of, where part_of(X, Y) is true if X is a descendant of Y in H_cell.

In the type of questions we addressed, at any given stage of the cell cycle process,
all cells in the experimental sample had the same number of nuclei; similarly for the other inner components. As a result, we could assume that, at every stage and for each link from a child \( X \) to its parent \( Y \) in \( H_{\text{cell}} \), this link was assigned a particular number indicating the number of elements of type \( X \) in one element of type \( Y \). The states of our domain were described by a basic fluent, \( \text{num} : \text{types_of_parts} \times \text{types_of_parts} \rightarrow \text{natural_numbers} \), where \( \text{num}(P_1, P_2) = N \) holds if the number of elements of type \( P_1 \) in one element of type \( P_2 \) is \( N \). For instance, \( \text{num}(\text{nucleus, cell}) = 2 \) indicates that, at the current stage of the cell cycle, each cell in the environment has two nuclei.

To describe the cell cycle we needed two action classes: \( \text{duplicate} \) and \( \text{split} \). \( \text{Duplicate} \), which acts upon an \textit{object} that is an element from sort \text{types_of_parts}, doubles the number of every part of this kind present in the environment. \( \text{Split} \) also acts upon an \textit{object} ranging over \text{types_of_parts}. An action \( a \) of this type with \( \text{object}(a) = c_1 \), where \( c_2 \) is a child of \( c_1 \) in \( H_{\text{cell}} \), duplicates the number of elements of type \( c_1 \) in the environment and cuts in half the number of elements of type \( c_2 \) in one element of type \( c_1 \). For example, if the experimental environment consists of one cell with two nuclei, the occurrence of an instance \( a \) of action \( \text{split} \) with \( \text{object}(a) = \text{cell} \) increases the number of cells to two and decreases the number of nuclei per cells to one, thus resulting in an environment consisting of two cells with only one nucleus each. In addition to these two actions we had an exogenous action, \( \text{prevent\_duplication} \), with an attribute \text{object} with the range \text{types_of_parts}. The occurrence of an instance action \( a \) of \( \text{prevent\_duplication} \) with \( \text{object}(a) = c \) nullifies the effects of duplication and splitting for the type \( c \) of parts. We made use of this exogenous action in representing external events that interfere with the normal succession of sub-phases of cell cycle. All this knowledge is represented by the following module:

```
module basic_cell_cycle

sort declarations
  types_of_parts :: universe
  attributes
    is_part_of :: types_of_parts

duplicate :: actions
  attributes
    object :: types_of_parts

split :: duplicate

prevent_duplication :: actions
  attributes
    object :: types_of_parts

function declarations
  statics
    defined
      part_of : types_of_parts \times types_of_parts \rightarrow \text{booleans}
```
fluen
total num : types_of_parts × types_of_parts → natural_numbers
prevented_dupl : types_of_parts → booleans

axioms

occurs(X) causes num(P_2, P_1) = N_2 if instance(X, duplicate),
object(X) = P_2,
is_part_of(P_2) = P_1,
num(P_2, P_1) = N_1,
N_1 * 2 = N_2.

occurs(X) causes num(P_2, P_1) = N_2 if instance(X, split),
object(X) = P_1,
is_part_of(P_2) = P_1,
num(P_2, P_1) = N_1,
N_2 * 2 = N_1.

occurs(X) causes prevented_dupl(P) if instance(X, prevent_duplication),
object(X) = P.

part_of(P_1, P_2) if is_part_of(P_1) = P_2.

num(P, P) = 0.
num(P_3, P_1) = N if is_part_of(P_3) = P_2,
p_1(P_3, P_2).

num(P_2, P_1) = N_1,
num(P_3, P_2) = N_2,
N_1 * N_2 = N.

impossible occurs(X) if instance(X, duplicate),
object(X) = P,
prevented_dupl(P).

Any model of cell cycle consists of a theory importing the two library modules presented above and a structure corresponding to the level of detail of that model. Let us consider a first model, in which we view cell cycle as a sequence consisting of interphase and the mitotic phase. This is represented in the structure by adding the attribute assignments component(1) = interphase and component(2) = mitotic_phase to the definition of instance cell_cycle. We remind the reader that such attribute assignments are read as “the 1st component of cell_cycle is interphase” and “the 2nd component of cell_cycle is mitotic_phase”. Interphase is considered an elementary action, while the mitotic phase splits the cell into two. We limit our domain to cells contained in an experimental environment, called sample.

system description cell_cycle(1)

theory
import module sequence from commonsense_lib
import module basic_cell_cycle from cell_cycle_lib
structure
instances
  sample in types_of_parts
cell in types_of_parts
  is_part_of = sample
cell_cycle in action_sequences
  length = 2
  component(1) = interphase
  component(2) = mitotic_phase
interphase in actions
mitotic_phase in split
  object = cell

This initial model of cell division is quite general. It was sufficient to answer a number of the questions targeted by the Digital Aristotle. There were, however, some questions which required a different model.

Consider, for instance, the following question from (Campbell and Reece 2001):

12.9. Text: In some organisms mitosis occurs without cytokinesis occurring.

Question: How many cells will there be in the sample at the end of the cell cycle, and how many nuclei will each cell contain?

To answer it, the system needed to know more about the structure of the cell and that of the mitotic phase. ALM facilitated the creation of a refinement of our original model of cell division: a new system description, cell_cycle(2), was easily created by adding to the previous structure a few new instances:

  nucleus in types_of_parts
  is_part_of = cell
mitosis in duplicate
  is_part_of = nucleus
cytokinesis in split
  is_part_of = cell

and replacing the old definition of the instance mitotic_phase by a new one:

  mitotic_phase in action_sequences
  length = 2
  component(1) = mitosis
  component(2) = cytokinesis

Similarly, various other refinements of our original model of cell division contained the same theory as the original formalization; only the structure of our original model needed to be modified, in an elaboration tolerant way. Matching questions with models of cell division containing just the right amount of detail is computationally advantageous and, in most cases, the matching can be done automatically.

Our formalization of cell division illustrates ALM’s capabilities of creating large knowledge bases for practical systems through its mechanisms for reusing knowledge. In our example, the two modules that formed the theory were directly im-
ported from the library into the system description. This shows that our main goal for $\mathcal{ALM}$ – the reuse of knowledge – was successfully achieved.

Additionally, the example demonstrates $\mathcal{ALM}$’s suitability for modeling not only commonsense dynamic systems, but also highly specialized, non-trivial domains. It shows the importance of creating and using libraries of knowledge in real-life applications, and it demonstrates the ease of elaborating initial formalizations of dynamic domains into more detailed ones.

Our second task in Project Halo was to develop a proof-of-concept question answering system that used $\mathcal{ALM}$ formalizations of cell cycle in solving complex temporal projection questions like 12.9 above. To do that, we used the methodology described in Section 4.2 expanded by capabilities for reasoning about naturally evolving processes. This latter part was done by incorporating a theory of intentions (Baral and Gelfond 2005) and assuming that naturally evolving processes have the tendency (or the intention) to go through their sequence of phases in order, unless interrupted (e.g., we can say that a cell tends/ intends to go through its cell cycle, which it does unless unexpected events happen).

In our question answering methodology, the structure of our $\mathcal{ALM}$ system description for the cell cycle domain provided the vocabulary for translating the questions expressed in natural language into a history. The theory of the system description contained the axioms encoding the background knowledge needed to answer questions about the domain.

As an example, the information given in the text of 12.9 above would be encoded by a history that contains the facts

\[
\begin{align*}
\text{observed} & (\text{num(cell, sample), 1, 0}) \\
\text{observed} & (\text{num(nucleus, cell), 1, 0}) \\
\text{intend} & (\text{cell\_cycle}, 0) \\
\neg\text{happened} & (\text{cytokinesis, I})
\end{align*}
\]

for every step $I$. Note that, unless otherwise specified, it would be assumed that the experimental sample consists of one cell with one nucleus.

The query in 12.9 would be encoded by the ASP\{f\} rules:

\[
\begin{align*}
\text{answer}(X, \text{“cells per sample”}) & \leftarrow \text{last\_step}(I), \\
& \quad \text{num(cell, sample, I)} = X. \\
\text{answer}(X, \text{“nuclei per cell”}) & \leftarrow \text{last\_step}(I), \\
& \quad \text{num(nucleus, cell, I)} = X.
\end{align*}
\]

Our system, $\mathcal{ALMAS}$, would solve the question answering problem by first generating a logic program consisting of the above facts and rules encoding the history and query, respectively; the ASP\{f\} translation of the $\mathcal{ALM}$ system description $\text{cell\_cycle}(2)$; and the temporal projection module described in Section 4.2. Then, the system would compute answer sets of this program, which correspond to answers to the question. For 12.9 there would be a unique answer set, containing:
intend(cytokinesis, 2)  ¬occurs(cytokinesis, 2)
intend(cytokinesis, 3)  ¬occurs(cytokinesis, 3)
intend(cytokinesis, 4)  ¬occurs(cytokinesis, 4)

These facts indicate that the unfulfillable intention of executing action cytokinesis persists forever. Additionally, the answer set would include atoms:

\[
\begin{align*}
\text{answer}(1, \text{“cells per sample”}) & \quad \text{holds}(\text{val}(\text{num(cell, sample)}, 1), 2) \\
\text{answer}(2, \text{“nuclei per cell”}) & \quad \text{holds}(\text{val}(\text{num(nucleus, sample)}, 2), 2) \\
\text{last_step}(2) & \quad \text{holds}(\text{val}(\text{num(nucleus, cell)}, 2), 2)
\end{align*}
\]

which indicate that at the end of the cell cycle there will be one cell in the sample, with two nuclei. This is in fact the correct answer to question 12.9.

This question answering methodology and the methodology of reasoning about naturally evolving processes using intentions was successfully applied to other questions about cell division.
Appendix C Comparison between Languages $\text{ALM}$ and $\text{MAD}$

In this section we give an informal discussion of the relationship between $\text{ALM}$ and the modular action language $\text{MAD}$ (Lifschitz and Ren 2006; Erdoğan and Lifschitz 2006). Both languages have similar goals but differ significantly in the proposed ways to achieve these goals. We believe that each language supports its own distinctive style of representing knowledge about actions and change. The difference starts with the non-modular languages that serve as the basis for $\text{ALM}$ and $\text{MAD}$. The former is a modular expansion of action language $\text{AL}$. The latter expands action language $\text{C}$ (Giunchiglia and Lifschitz 1998). Even though these languages have a lot in common (see (Gelfond and Lifschitz 2012)) they differ significantly in the underlying assumptions incorporated in their semantics. For example, the semantics of $\text{AL}$ incorporates the Inertia Axiom, which says that “Things normally stay the same.” Language $\text{C}$ is based on a different assumption – the Causality Principle – which says that “Everything true in the world must be caused.” Its underlying logical basis is causal logic (McCain and Turner 1997; Giunchiglia et al. 2004a). In $\text{C}$ the inertia axiom for a literal $l$ is expressed by a statement

\[
\text{caused } l \text{ if } l \text{ after } l, 
\]

read as “there is a cause for $l$ to hold after a transition if $l$ holds both before and after the transition”. While $\text{AL}$ allows two types of fluents – inertial and defined $\sim$, $\text{C}$ can be used to define other types of fluents (e.g., default fluents that, unless otherwise stated, take on the fixed default values). The authors of this paper did not find these types of fluents to be particularly useful and, in accordance with their minimalist methodology, did not allow them in either $\text{AL}$ or $\text{ALM}$. Of course, the question is not settled and our opinion can change with additional experience. On another hand, $\text{AL}$ allows recursive state constraints and definitions, which are severely limited in $\text{C}$. There is a close relationship between ASP and $\text{C}$ but, in our judgment, the distance between ASP and $\text{AL}$ is smaller than that between ASP and $\text{C}$. There is also a substantial difference between modules of $\text{ALM}$ and $\text{MAD}$.

To better understand the relationship let us consider the $\text{ALM}$ theory $\text{motion}$ and the system description $\text{travel}$ from Section 3.2 and represent them in $\text{MAD}$

Example 12 (A MAD Version of the System Description travel)

The $\text{ALM}$ system description $\text{travel}$ is formed by the theory $\text{motion}$ and the structure $\text{Bob and John}$. The theory consists of two modules, $\text{moving}$ and $\text{carrying things}$, organized into a module hierarchy in which the latter module depends on the former. Let us start with the $\text{MAD}$ representation of $\text{ALM}$’s module $\text{moving}$.

In general, the representation of an $\text{ALM}$ module $M$ in $\text{MAD}$ consists of two parts: the declaration of sorts of $M$ and their inclusion relation, and the collection of $\text{MAD}$ modules corresponding to $M$. (In our first example a module of $\text{ALM}$ will be

---

12 Although the “Monkey and Banana” problem presented in Section 4.1 has been encoded in $\text{MAD}$ as well (Erdoğan 2008), we are not considering it here because of the length of its representation and, most importantly, because there are substantial differences in how the problem was addressed in $\text{ALM}$ versus $\text{MAD}$ from the knowledge representation point of view.
mapped into a single module of MAD.) Note that sorts can also be declared within the module but in this case they will be local (i.e., invisible to other modules). Declarations given outside of a module can be viewed as global.

In our case, the sorts and inclusions sections of the translation $M_1 = \text{MAD}(\text{moving})$ consist of the following statements (We remind the reader that in MAD variables are identifiers starting with a lower-case letter and constants are identifiers starting with an upper-case letter, the opposite of $\text{ALM}$):

**sorts**
- Universe; Points; Things; Agents;

**inclusions**
- Points $\ll$ Universe;
- Things $\ll$ Universe;
- Agents $\ll$ Things;

The sorts part declares the sort universe (which is pre-defined and does not require declaration in $\text{ALM}$) together with the sorts of moving that are not special cases of actions. The inclusions part describes the specialization relations between these sorts. The definition of a MAD module starts with a title:

**module** $M_1$

The body of a MAD module consists of separate (optional) sections for the declarations of sorts specific to the current module, objects, fluents, actions, and variables, in this order, together with a section dedicated to axioms (Erdogan 2008). Our module $M_1$ starts with the declarations of fluents:

**fluents**
- Symmetric_connectivity : rigid;
- Transitive_connectivity : rigid;
- Connected(Points, Points) : simple;
- Loc_in(Things) : simple(Points);

Rigid fluents of MAD are basic statics of $\text{ALM}$.

To declare the action class move of moving we need to model its attributes. To do that we introduce variables with the same names as the associated attributes in moving. This will facilitate referring to those attributes later in axioms. We also order attributes alphabetically as arguments of the action term to ease the translation of special case action classes of move:

**actions**
- Move(Agents, Points, Points);

The variable declaration and axiom part come next. We will need to add extra axioms (and associated variables) to say that Loc_in is an inertial fluent (i.e., basic fluent in $\text{ALM}$ terminology) and that Move(Agents, Points, Points) is an exogenous action (i.e., it does not need a cause in order to occur; it may or may not occur at any point in time).
variables
  t, t₁, t₂ : Things;
  actor : Agents;
  origin, dest : Points;
axioms
  inertial Loc_in(t);
  exogenous Move(actor, dest, origin);

The causal law for move can now be expressed in a natural way:

\[ Move(actor, dest, origin) \text{ causes } Loc_{in}(actor) = dest; \]

Similarly for the executability conditions:

**nonexecutable** Move(actor, dest, origin) \text{ if } Loc_{in}(actor) \neq \text{origin};

**nonexecutable** Move(actor, dest, origin) \text{ if } Loc_{in}(actor) = \text{dest};

**nonexecutable** Move(actor, dest, origin) \text{ if } Loc_{in}(actor) = \text{origin},
\neg \text{Connected(origin, dest)};

The situation becomes substantially more difficult for the definition of Connected. The definition used in moving is recursive and therefore cannot be easily emulated by MAD’s causal laws. The relation can, of course, be explicitly specified later together with the description of particular places, but this causes considerable inconvenience.

To represent module carrying_things from the theory motion we need a new (global) sort:

**sorts** Carriables;

**inclusions** Carriables ≪ Things;

The module M₂ that corresponds to carrying_things contains declarations of the new action Carry and the corresponding variables.

**module M₂**;

**actions**
  Carry(Agents, Carriables, Points, Points);

**variables**
  t : Things;
  actor : Agents;
  dest, origin, p : Points;
  carried_object, c : Carriables;

Next we need to define axioms of the module. Clearly we need to say that the action Carry(actor, carried_object, dest, origin) is a special case of the action Move(actor, dest, origin). Since ALM allows action sorts, no new mechanism is required to do that in carrying_things. In MAD, while there is a built-in sort action, special case actions are not sorts and the special constructs import and is are
introduced to achieve this goal. Special case actions are declared in MAD by importing the module containing the original action and renaming the original action as the special case action as follows:

\[
\text{import } M_1; \\
\text{Move}(\text{actor}, \text{dest}, \text{origin}) \text{ is } \text{Carry}(\text{actor}, \text{carried} \text{object}, \text{dest}, \text{origin});
\]

Intuitively, this import statement says that the action \(\text{Carry}(\text{actor}, \text{carried} \text{object}, \text{dest}, \text{origin})\) has all properties that are postulated for the action \(\text{Move}(\text{actor}, \text{dest}, \text{origin})\) in the module \(M_1\). We also need an additional axiom declaring the action to be exogenous, and state constraints, and executability conditions similar to those in \(\text{carrying} \text{things}\):

\[
\text{axioms}
\]

\[
\text{exogenous } \text{Carry}(\text{actor}, \text{carried} \text{object}, \text{dest}, \text{origin});
\]

\[
\% \text{ State constraints:}
\text{Is}_\text{held}(c) \quad \text{if} \quad \text{Holding}(t, c);
\]

\[
\% \text{ Executability conditions:}
\text{nonexecutable } \text{Carry}(\text{actor}, \text{carried} \text{object}, \text{dest}, \text{origin}) \text{ if } \neg \text{Holding}(\text{actor}, \text{carried} \text{object});
\]

\[
\text{nonexecutable } \text{Move}(\text{actor}, \text{dest}, \text{origin}) \text{ if } \text{Is}_\text{held}(\text{actor});
\]

Note, however, that the \(\text{ALM}\) module \(\text{carrying} \text{things}\) also contained the recursive state constraints below, saying that agents and the objects they are holding have the same location:

\[
\text{loc}_\text{in}(C) = P \quad \text{if} \quad \text{holding}(T, C), \text{loc}_\text{in}(T) = P; \\
\text{loc}_\text{in}(T) = P \quad \text{if} \quad \text{holding}(T, C), \text{loc}_\text{in}(C) = P.
\]

Since this is not allowed in MAD, we have to use a less elaboration tolerant representation by adding an explicit causal law saying

\[
\text{Move}(\text{actor}, \text{dest}, \text{origin}) \text{ causes } \text{Loc}_\text{in}(c) = \text{dest} \text{ if } \text{Holding}(\text{actor}, c);
\]

In MAD additional axioms will be needed to rule out certain initial situations (e.g., “John is holding his suitcase. He is in Paris. His suitcase is in Rome.”) or to represent and reason correctly about more complex scenarios (e.g., “Alice is in the kitchen, holding her baby who is holding a toy. Alice goes to the living room.”).

This completes the construction of \(M_2\).

In general, special case actions are declared in MAD by importing the module containing the original action and renaming the original action as the special case action. That is why we needed to place the MAD representation of \(\text{carry}\) in a new module that we call \(M_2\), in which we import module \(M_1\) while renaming \(\text{Move}(\text{actor}, \text{dest}, \text{origin})\) as \(\text{Carry}(\text{actor}, \text{carried} \text{object}, \text{dest}, \text{origin})\). In \(\text{ALM}\) the declarations of \(\text{move}\) and its specialization \(\text{carry}\) could be placed in the same module – the decision is up to the user – whereas in MAD they must be placed in separate modules. This potentially leads to a larger number of smaller modules in MAD than in \(\text{ALM}\) representations.

Finally, we consider the structure of our \(\text{ALM}\) system description. It contains two types of actions \(\text{go}(\text{Actor}, \text{Dest})\) and \(\text{go}(\text{Actor}, \text{Dest}, \text{Origin})\). Let us expand the
structure by a new object, *suitcase*, and a new action *carry(Actor, suitcase, Dest)*. For illustrative purposes, let us assume that we would like the MAD representation to preserve these names.

To represent this in MAD, we introduce a new module *S*. It has the local definitions of objects:

```
module S;
  objects
    John, Bob : Agents;
    New_York, Paris, Rome : Points;
    Suitcase : Carriables;
```

and those of actions. The latter can be defined via the renaming mechanism of MAD. This requires importing the modules in which the action classes were declared. Thus, module *S* imports modules *M*₁ and *M*₂.

```
actions
  Go(Agents, Points);
  Go(Agents, Points, Points);
  Carry(Agents, Carriables, Points);
variables
  actor : Agents;
  origin, dest : Points;
import M₁;
  Move(actor, dest, origin) is Go(actor, dest, origin);
import M₁;
  Move(actor, dest, origin) is Go(actor, dest);
import M₂;
  Carry(actor, Suitcase, dest, origin) is Carry(actor, Suitcase, dest)
```

This completes the construction of the MAD representation of the system description *travel*.

Even this simple example allows to illustrate some important differences between *ALM* and MAD. Here is a short summary:

- **Recursive definitions**
  The representation of state constraints of an *ALM* system description is not straightforward if the set of state constraints defines a cyclic fluent dependency graph (Gelfond and Lifschitz 2012). For instance, the *ALM* state constraint:

  \[ p \text{ if } p. \]

  is not equivalent to the same axiom in MAD. The *ALM* axiom can be eliminated without modifying the meaning of the system description; it says that “in every state in which \( p \) holds, \( p \) must hold.” Eliminating the same axiom from a MAD action description would not produce an equivalent action description; in MAD, the axiom says that “\( p \) holds by default.” This difference
between $\mathcal{ALM}$ and $\text{MAD}$ is inherited from the similar difference between $\mathcal{AL}$ and $\mathcal{C}$.

- **Separation of Sorts and Instances**
  One of the most important features of $\mathcal{ALM}$ is its support for a clear separation of the definition of sorts of objects of the domain (given in the system’s theory) from the definition of instances of these sorts (given by the system’s structure). Even though it may be tempting to view the first two modules, $M_1$ and $M_2$ above as a $\text{MAD}$ counterpart of the $\mathcal{ALM}$ theory $\text{motion}$, the analogy does not hold. Unlike $\mathcal{ALM}$ where the corresponding theory has a clear semantics independent of that of the structure, no such semantics exists in $\text{MAD}$. Modules $M_1$ and $M_2$ only acquire their meaning after the addition of module $S$ that corresponds to the $\mathcal{ALM}$’s structure. We believe that the existence of the independent semantics of $\mathcal{ALM}$ theories facilitates the stepwise development and testing of the knowledge base and improves their elaboration tolerance.

- **Action Sorts**
  In $\mathcal{ALM}$, the pre-defined sort $\text{actions}$ is part of the sort hierarchy, whereas in $\text{MAD}$ actions are not considered sorts. Instead, $\text{MAD}$ has special constructs $\text{import}$ and $\text{is}$ (also known as bridge rules), which are used to define actions as special cases of other actions. No such special constructs are needed in $\mathcal{ALM}$.
  Moreover, in $\mathcal{ALM}$, an action class and its specialization can be part of the same module. This is not the case in $\text{MAD}$ where a special case of an action class must be declared in a separate module by importing the module containing the original action class and using renaming clauses. As a consequence, the $\text{MAD}$ representation of $\mathcal{ALM}$ system descriptions will generally contain more modules that are smaller in size than the $\mathcal{ALM}$ counterpart. On the other hand, note that $\mathcal{ALM}$ modules are not required to be large; they can be as small as a user desires.
  $\mathcal{ALM}$ allows the definition of fluents on (or ranging over) specific action classes only, and not necessarily the whole pre-defined $\text{actions}$ sort, for instance:
  
  $\text{intended : agent\_actions} \rightarrow \text{booleans}$

  where $\text{agent\_actions}$ is a special case of $\text{actions}$. There is no equivalent concept in $\text{MAD}$, where fluents must be defined on, and range over, either primitive sorts or the built-in sort $\text{action}$, but not specific actions.

- **Variable Declarations**
  In $\mathcal{ALM}$, we do not define the sorts of variables used in the axioms. This information is evident from the atoms in which they appear. In $\text{MAD}$, variables need to be defined, which may lead to larger modules and cause errors related to use of variables of wrong types.

- **Renaming Feature of $\text{MAD}$**
  In $\text{MAD}$, sorts can be renamed by importing the module containing the original declaration of a sort and using a renaming clause. The meaning of such a
renaming clause is that the two sorts are synonyms. There is no straightforward way to define this synonymy in ALM. The closest thing is to use the specialization construct of our language and declare the new sort as a special case of the original one. The reverse (i.e., the original sort being a special case of the renamed sort) cannot be added, as sort hierarchies of ALM are required to be DAGs. This leads to further problems when the renamed sorts appear as attributes in renamed actions of MAD.

- **Axioms of MAD that have no equivalent in ALM**

Some axioms, allowed in MAD, are not directly expressible in ALM. For instance, MAD axioms of the type:

\[
\text{formula may cause formula [ if formula ]}
\]

or

\[
\text{default formula [ if formula ] [ after formula ]}
\]

belong to this group. The first axiom allows to specify non-deterministic effects of actions, while the second assigns default values to fluents (and more complex formulas). As discussed above, we are not yet convinced that the latter type of axioms needs to be allowed in ALM. Non-determinism, however, is an important feature that one should be able to express in an action formalism. It may be added to ALM (and to ALC) in a very natural manner, but it is not allowed in ALC and the mathematical properties of “non-deterministic” ALC were not yet investigated. Because of this we decided to add this feature in the next version of ALM.

We hope that this section gives the reader some useful insight in differences between ALM and MAD. We plan to extend the comparison between ALM and MAD in the future. Formally investigating the relationship between the two languages can facilitate the translation of knowledge modules from one language to another, and can identify situations when one language is preferable to the other. Readers interested in a formal translation of system descriptions of ALM to action descriptions of MAD can consult [Incezan 2012].

**References**


Balduccini, M. 2004. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL’04. Lecture Notes in Artificial Intelligence (LNCS).*


Daniela Inclezan and Michael Gelfond


