A CLIB-Inspired Library of Commonsense Knowledge in Modular Action Language $\mathcal{ALM}$

Daniela Inclezan
Miami University
Oxford, Ohio 45056
inclezd@MiamiOH.edu

Abstract

This paper describes a modular action language, $\mathcal{ALM}$, dedicated to the specification of complex dynamic systems. One of the main goals of the language is to facilitate the development and testing of knowledge representation libraries. We present the implementation of a large scale library of commonsense concepts, achieved by porting knowledge from the *Component Library* (CLIB) into $\mathcal{ALM}$. Our choice of CLIB as a source of inspiration is justified by the well-founded methodology used by its authors in selecting the general concepts it contains, and its extensive testing in the context of the Automated User-centered Reasoning and Acquisition System. The resulting $\mathcal{ALM}$ library has the additional advantage of incorporating established knowledge representation methodologies developed in the *action language* research community.

Introduction

This paper describes a modular action language, $\mathcal{ALM}$ (Action Language with Modules), dedicated to the specification of complex dynamic systems, and presents the implementation of an $\mathcal{ALM}$ library of commonsense concepts, achieved by translating knowledge from the *Component Library* (CLIB) (Barker, Porter, and Clark 2001) into $\mathcal{ALM}$.

Dynamic systems that change because of actions and evolve in discrete steps (i.e., *discrete dynamic systems*) can theoretically be modeled by transition diagrams whose nodes represent physical states of the domain and whose arcs are labeled by actions. *Action languages* (Gelfond and Lifschitz 1998) were introduced as one of the solutions to the long debated problem of finding the concise and mathematically accurate description of transition diagrams. These are formal languages that describe the effects of actions and action preconditions in a syntax close to that of natural language. Several action languages exist nowadays. They incorporate, in various degrees, solutions to important problems from the field of representing and reasoning about actions (e.g., the *frame*, *ramification*, or *qualification* problems). Action languages sometimes differ in their underlying assumptions. For instance, the semantics of action language $\mathcal{AL}$ (Turner 1997; Baral and Gelfond 2000) relies on the Inertia Axiom (McCarthy and Hayes 1969), which expresses the intuition that *"Normally things stay the same"*.

Although traditional action languages represent a substantial advancement, they are unsuitable for representing knowledge about *large* dynamic systems. First of all, they lack the means for structuring knowledge and representing hierarchies of abstraction, relevant for the design of knowledge bases and the creation of libraries. Secondly, traditional action languages do not thoroughly address the issue of how to represent objects of the domain (including actions). They do not possess means for describing *optional* attributes of objects, or objects as special cases of other objects—a common practice in natural language (e.g., "*carry*" is defined as "*move while holding*," a special case of "*move*").

Modular action language $\mathcal{ALM}$ intends to remedy these problems and allow for the elaboration tolerant representation of knowledge about large dynamic systems. $\mathcal{ALM}$ incorporates the underlying ideas of $\mathcal{AL}$, and addresses the issues of the traditional approach by (1) separating a general *theory* from its *structure*; (2) organizing knowledge into *modules*; (3) introducing *classes* of objects with *optional* attributes; and (4) allowing the declaration of classes in terms of previously defined ones. Other modular action languages exist, but they have different underlying assumptions. For instance, modular language MAD (Lifschitz and Ren 2006; Erdoğan and Lifschitz 2006) is based on the Causality Principle that says that "*everything true in the world must be caused*" (McCain and Turner 1997; Giunchiglia and Lifschitz 1998; Giunchiglia et al. 2004).

A first version of our language was introduced in (Gelfond and Inclezan 2009). Since then, substantial improvements have been made based on our practice in formalizing knowledge. As an example, we tested our language in the context of Project Halo (Gunning et al. 2010) by representing specialized knowledge about a biological process (Inclezan and Gelfond 2011).

After defining our language, our next goal was to develop an $\mathcal{ALM}$ library of commonsense knowledge that would facilitate the description of large dynamic systems through the reuse of its components. We started by creating a small library of motion. We were satisfied with the result, and were able to use our representation in solving reasoning tasks by combining system descriptions of $\mathcal{ALM}$ with reasoning algorithms written in Answer Set Prolog (Gelfond and Lifschitz 1988; 1991). However, we were not sure what criteria to use for the selection of further concepts, general enough to deserve inclusion in our commonsense library.

An answer to our question was provided by our collaboration on project AURA (Automated User-centered Reasoning and Acquisition System) (Chaudhri et al. 2007; Clark et al. 2007; Chaudhri et al. 2009). AURA is a knowledge acquisition system that allows domain experts to enter knowledge and questions related to different disciplines, with minimum support from knowledge engineers. Our task within the project was to revise a section of AURA's core knowledge base, called *Component Library* (CLIB), from the point of view of its soundness and completeness with respect to the goals of AURA (Chaudhri, Dinesh, and Inclezan 2014). CLIB (Barker, Porter, and Clark 2001) is a vast library of general, reusable knowledge components with goals similar to ours. It was extensively tested in AURA along the years and, more importantly for us, its concepts were selected using a well-founded methodology, based on lexical and ontological resources. We benefited from these key features of CLIB by porting the library, in its revised form resulting from our analysis, into $\mathcal{ALM}$. The resulting $\mathcal{ALM}$ library combines the advantages of CLIB with those of incorporating established knowledge representation methodologies developed in the *action language* community. The process of creating a CLIB-inspired library in $\mathcal{ALM}$ has also allowed us to find a few areas of future refinement of our language and the methodology of its use.

In what follows we describe language $\mathcal{ALM}$. We present CLIB and AURA, and introduce our CLIB-inspired library in $\mathcal{ALM}$. We end with conclusions and future work.

Modular Action Language $\mathcal{ALM}$

Syntax

A dynamic system is represented in $\mathcal{ALM}$ by a *system description* that consists of two parts: a *general theory* (i.e., a collection of modules with a common theme organized in a hierarchy) and a *structure* (i.e., an interpretation, in the classical logic sense, of some of the symbols in the theory). A *module* is a collection of declarations of classes and functions together with a set of axioms. The purpose of a module is to allow the organization of knowledge into smaller reusable pieces of code – modules serve a similar role to that of procedures in procedural languages.

We briefly illustrate the syntax of $\mathcal{ALM}$ via some examples. Note that boldface symbols denote keywords of the language; identifiers starting with a lowercase letter denote constant symbols; and identifiers starting with a capital letter denote variables.

Let us consider a dynamic system in which we have $things$ and discrete $points$ in space. Certain things, called $movers$, are able to move from one point to another. We are interested in the locations of $things$, as well as the effects of action $move$ and its executability conditions. To describe this domain, we create a module called $basic\_motion$ that we declare as follows:

$\quad$ **module** $basic\_motion$

Then, we declare the classes of objects in our domain, which are organized into a DAG class hierarchy with the root $universe$; $points$, $things$, and $actions$ are children of $universe$; $movers$ is a child of $things$; and $move$ is a child of $actions$. Note that in our language $universe$ is a predefined class and the root of every class hierarchy.

class declarations
$\quad points, things :: universe$
$\quad movers :: things$

The above declarations say that $things$, $points$, and $movers$ are classes, where $movers$ is a special case of $things$. The symbol $::$ above denotes the *specialization* construct, introduced to represent links between nodes in the class hierarchy. Multiple links can be represented in a single statement, as shown in the declaration of classes $points$ and $things$.

The only class remaining to declare is $move$, which is a subclass of the pre-defined class $actions$ of our language; $move$ has three *attributes* (i.e., three intrinsic properties): $actor$, $origin$, and $destination$ (shortened as $dest$). These are possibly partial functions that map elements of class $move$ into elements of class $movers$, $points$, and $points$, respectively. For readability purposes, we do not repeat the domain $move$ of these attributes and simply write:

$\quad move :: actions$
$\quad\quad$ **attributes**
$\quad\quad\quad actor : movers$
$\quad\quad\quad origin : points$
$\quad\quad\quad dest : points$

The next component of our module is the declaration of functions in our domain. Functions represent relevant relations between domain objects, and are divided into $statics$ (those properties that cannot be changed by actions) and $fluents$ (those that can). Statics and fluents are further divided into $basic$ and $defined$, where defined functions are described in terms of other functions and can be viewed as shorthands used for the ease of representation. In our $basic\_motion$ domain, the only property of interest is the location of things, which can be changed by actions of type $move$ – hence it is a *fluent*; it is also a *basic* and *total* fluent, as we will not define it in terms of other functions and we assume that the location of all things is known at each step in time. We encode all this information in $\mathcal{ALM}$ as follows:

function declarations
$\quad$ **fluents**
$\quad\quad$ **basic**
$\quad\quad\quad$ **total** $loc\_in : things \rightarrow points$

The final part of a module contains axioms that describe direct effects of actions (*dynamic causal laws*), indirect effect of actions (*state constraints*), definitions of defined functions (*function definitions*), and conditions for the executability of actions (*executability conditions*). This section starts with the keyword

axioms

followed by domain axioms ending with a period symbol (.). In our scenario, we have a dynamic causal law specifying that the actor of a $move$ action will be located at the destination after the execution of the action:

$$\begin{array}{lll} occurs(X) & \textbf{causes} & loc\_in(A) = D \\ & \textbf{if} & instance(X, move), \\ & & occurs(X), \\ & & actor(X) = A, \\ & & dest(X) = D. \end{array}$$

We also have two executability conditions, one of them specifying that a $move$ action cannot occur if the actor is not located at the origin:

$$\begin{array}{ll} \textbf{impossible} & occurs(X) \\ \textbf{if} & instance(X, move), \\ & actor(X) = A, \\ & origin(X) = O, \\ & loc\_in(A) \neq O. \end{array}$$

and the other preventing circular movements in which the destination is the same as the initial location of the actor:

$$\begin{array}{ll} \textbf{impossible} & occurs(X) \\ \textbf{if} & instance(X, move), \\ & actor(X) = A, \\ & dest(X) = D, \\ & loc\_in(A) = D. \end{array}$$

Next, let us illustrate how classes of $\mathcal{ALM}$ can be defined in terms of other classes. Let us imagine that some of the objects of our domain are light enough to be carried between $points$ by $movers$. We call such objects $carriables$. In the dictionary, action $carry$ is defined as *"to move while supporting."* To encode this information, we expand the module $basic\_motion$ by the following class declarations:

$carriables$ :: $things$
$carry$ :: $move$
attributes
$carried\_thing : carriables$

This says that $carry$ is a special case of the action class $move$, meaning that, in addition to its own attributes and axioms, $carry$ inherits the attributes and axioms of $move$.

Our module will also contain the following basic total fluent declaration, which will be added underneath the declaration of loc_in shown above:

total $supports : things \times carriables \rightarrow booleans$

(We assume that $booleans$ is a pre-defined and pre-interpreted class of our language).

We expand the axioms of the original module by the following state constraint saying that the location of a supported thing is the same as the location of its supporter:

$$loc\_in(C) = loc\_in(T) \quad \textbf{if} \quad supports(T, C).$$

and the executability condition:

$$\begin{array}{ll} \textbf{impossible} & occurs(X) \\ \textbf{if} & instance(X, carry), \\ & actor(X) = A, \\ & carried\_thing(X) = C, \\ & \neg supports(A, C). \end{array}$$

that says that only the supporter of a thing can carry it around.

New modules of knowledge can be developed and tested independently from existing ones, while reusing already encoded information from the older modules. This is done via the *module dependency* syntactic construct that allows knowledge engineers to create tree-like hierarchies of modules in which declarations and axioms from parent modules do not have to be explicitly repeated in children modules, as they are considered implicit. A collection of modules satisfying certain restrictions forms a *theory* of $\mathcal{ALM}$. Intuitively, a theory describes a collection of transition diagrams corresponding to different scenarios that may differ in the instantiation of classes of the theory or the values of statics.

Theories containing very general information can be stored into *libraries* and can be imported from there into system descriptions using *import* statements. For instance, imagine that the general knowledge about motion represented above is stored in a theory called $commonsense\_motion$, which currently consists of only the $basic\_motion$ module, and that this theory is part of a library named $commonsense\_lib$.

The second part of a system description, specified after the theory, is its *structure*. The structure represents a specific scenario for the domain described in the theory. For example, let us consider a scenario of the motion domain described earlier, in which two people, John and Bob, travel between London and Paris; Bob may carry his suitcase with him. To encode this scenario, we start with the keyword **structure** and a name, and continue with the definitions of instances of our classes:

structure $travel$
instances
$john, bob$ **in** $movers$
$london, paris$ **in** $points$
$suitcase$ **in** $carriables$
$move(A, P)$ **in** $move$
$actor = A$
$dest = P$
$bob\_suitcase\_lp\_carry$ **in** $carry$
$actor = bob$
$carried\_thing = suitcase$
$origin = london$
$dest = paris$

The definition of $move(A, P)$ above, called an *instance schema*, is a shorthand for a set of instances that includes, for example, the instance:

$move(john, london)$

with the following values assigned to its attributes:

$actor = john$
$dest = london$

Our particular domain is described by a system description whose theory, $commonsense\_motion$, is imported from the $commonsense\_lib$ library and whose structure is the one presented earlier:

system description $travel$
import $commonsense\_motion$
from $commonsense\_lib$
structure $travel$
$\langle structure\ body \rangle$

This concludes the brief exemplification of the syntax of $\mathcal{ALM}$.

Informal Semantics

Semantically, a theory can be viewed as a function that maps possible interpretations of its symbols into transition diagrams describing dynamic domains. We give the semantics of $\mathcal{ALM}$ by defining the states and transitions of the transition diagram defined by a system description. For that purpose, we encode statements of the system description into a logic program of ASP{f} (Balduccini 2012), an extension of Answer Set Prolog (Gelfond and Lifschitz 1991) by non-Herbrand functions. The states and transitions of the corresponding transition diagram will be determined by parts of the answer sets of this logic program. As an example, the dynamic causal law about actions of the type $move$ shown earlier is encoded as the following ASP{f} rule:

$$\begin{aligned} loc\_in(A, I+1) = D \quad \leftarrow \quad & instance(X, move), \\ & occurs(X, I), \\ & actor(X) = A, \\ & dest(X) = D. \end{aligned}$$

The remainder of the theory is translated in an equally simple manner. The structure is encoded using statements like:

$$instance(bob, movers).$$

$$\begin{aligned} instance(move(A, P), move) \quad \leftarrow & \\ & instance(A, movers), \\ & instance(P, points). \end{aligned}$$

Note that the encoding of a theory in ASP{f} needs to be done only once. Then, this encoding can be used together with the encodings of any number of its structures. This illustrates the reuse of knowledge in $\mathcal{ALM}$.

System descriptions of $\mathcal{ALM}$ are normally used in conjunction with the description of the system's history (Balduccini and Gelfond 2003) to perform a variety of reasoning tasks like temporal projection, diagnosis, or reasoning about process interruptions (Inclezan and Gelfond 2011).

CLIB and AURA

CLIB (Barker, Porter, and Clark 2001) is an ontology of general, reusable, composable, and interrelated components of knowledge. One of its goals is to provide domain experts with means for encoding knowledge from their fields, with no or minimal involvement from the part of knowledge engineers. Notions included in CLIB were selected using a solid methodology relying on linguistic and ontological resources such as WordNet,[1] FrameNet,[2] VerbNet,[3] a thesaurus and an English dictionary, as well as various ontologies from the semantic web community. CLIB was built with three main design criteria in mind: (1) *coverage*: CLIB should contain enough components to allow representing a variety of knowledge; (2) *access*: components should meet users' intuition and be easy to find; and (3) *semantics*: components should be enriched with non-trivial axioms.

The library is written in the knowledge representation language of Knowledge Machine (KM) (Clark and Porter 2004). KM is a frame-based language with first-order logic semantics. It has two types of basic concepts: *class* and *instance*. A *slot* is a special type of a class that defines a relation between two classes (it sometimes corresponds to an attribute of a class in $\mathcal{ALM}$). CLIB organizes its components into three main classes: *entities*, *events*, and *roles*. Events are divided into *actions* and *states*. An example of a CLIB state is `Be-Touching`, which would be called a *domain property* in action language terminology. A state of CLIB should not be confused with a state of a transition diagram. To distinguish between the two, we will use the expression "CLIB-state" for the former and "state" for the latter. The CLIB library was integrated in two systems developed at SRI International, SHAKEN and its successor AURA, and was extensively tested as a result.

AURA (Automated User-centered Reasoning and Acquisition System) (Chaudhri et al. 2007; Clark et al. 2007) is a knowledge-based system whose goal is "to enable domain experts to construct declarative knowledge bases from parts of a science textbook for Physics, Chemistry, and Biology in a way that another user can pose questions similar to those in an Advanced Placement exam and get answers and explanations" (Chaudhri et al. 2009). In AURA, domain experts encode new knowledge by building upon the general concepts of CLIB: they create directed graph structures specifying relations between new and existing components in a user-friendly graphical interface. Tests performed on the system showed promising results: knowledge was captured in a speedy manner and at least 70% of the questions from an Advanced Placement test suite were accurately answered by the AURA system in all three domains of interest (Gunning et al. 2010). These results seem to demonstrate that CLIB is a valuable library of general concepts. On the other hand, the extensive use of CLIB by knowledge engineers also indicated aspects of the library that needed revision. The author of this article was involved in the activity of reviewing and refining part of CLIB based on the lessons learned from its use in AURA (Chaudhri, Dinesh, and Inclezan 2014).

As a second task within project AURA, we translated CLIB into modular action language $\mathcal{ALM}$. The goal was to create a CLIB-inspired knowledge base that would take advantage of the methodologies developed in the *action language* community for the representation of knowledge and reasoning about dynamic systems (e.g., solutions to the frame and ramification problems). $\mathcal{ALM}$ is a good choice for such an endeavor as its relative closeness to the object-oriented paradigm (in comparison with more traditional action languages) eases the translation from KM. More importantly, encoding CLIB in $\mathcal{ALM}$ leads to an improved representation of CLIB action classes, as detailed below:

1. Axioms are specified in $\mathcal{ALM}$ in an elaboration tolerant way, and in a syntax close to that of natural language, whereas in KM they are described using STRIPS-like operators (e.g., add lists, delete lists, etc.) and a more complex syntax at times.

2. System descriptions of $\mathcal{ALM}$ can be used to solve various computational tasks by coupling them with reasoning algorithms written in ASP or its extensions (e.g., ASP{f} (Balduccini 2012), CR-Prolog (Balduccini and Gelfond 2003)). On the other hand, the inference engine of KM cannot perform planning nor postdiction (i.e., projection backwards in time when the present is not completely known). As a result, the $\mathcal{ALM}$ version of CLIB could be used to answer more complex questions about the targeted scientific domains, for instance questions that re-

[1] wordnet.princeton.edu/

[2] framenet.icsi.berkeley.edu/

[3] verbs.colorado.edu/verb-index/

quire finding a diagnostic (e.g., "*A sample cell was situated in a medium favorable to cell division. However, cell division did not occur. What can be the explanation?*").

A CLib-Inspired $\mathcal{ALM}$ Library

Our $\mathcal{ALM}$ translation focused mainly on the 147 actions of CLib, but some related concepts, such as CLib-states, had to be addressed as well. The result was an $\mathcal{ALM}$ library called *clib* consisting of eight theories. In what follows, we describe some of the challenges we encountered in constructing this library and show more details about one of the produced $\mathcal{ALM}$ modules, contained by the $motion$ theory of this new library.

Challenges

In porting the CLib library into $\mathcal{ALM}$ we had two main concerns:

1) How to translate entities, events, and roles of CLib into classes, functions, and axioms of $\mathcal{ALM}$.
2) How to conveniently group the resulting classes, functions, and axioms into modules of $\mathcal{ALM}$.

In addressing the first concern, we faced several difficulties, most of them derived from the differences in basic concepts between KM and our language:

- The two languages share the basic concepts of *class* and *instance*, but $\mathcal{ALM}$ has an extra concept, that of a *function*. Properties of dynamic systems (i.e., domain properties: statics and fluents) are described in $\mathcal{ALM}$ by functions. On the other hand, CLib-states, which correspond roughly to domain properties, are represented in CLib by classes, and their parameters and range are encoded by attributes of these classes, where attributes may be optional. For instance, the declaration of the CLib-state `Be-Accessibile` looks as follows:

```
(Be-Accessible has
  (superclasses (State)))
(every Be-Accessible has
  (object ((a Entity)))
  (base ((must-be-a Thing)))       ...
```

  This says that a `Be-Accessible` CLib-state has an `object` that is required and must be of type `Entity`, and a `base` that is optional, but if it exists it must be of type `Thing`. If we think in terms of functions, `Be-Accessible` is a boolean function with a variable arity, either 1 or 2. The translation challenge is that in $\mathcal{ALM}$ the number and order of a function's parameters is fixed. To solve this problem, multiple domain properties are defined in $\mathcal{ALM}$ for each CLib-state with optional parameters, to reflect the various combinations of required and optional parameters. For instance, the CLib-state `Be-Accessible` is encoded in $\mathcal{ALM}$ using two domain properties, declared as follows:

  fluents
  basic $accessible\_to : thing \times entity \rightarrow booleans$
  defined $accessible : entity \rightarrow booleans$

  The domain property $accessible$ will be defined in terms of $accessible\_to$ via state constraints.

- As CLib-states are specified by classes and classes can be organized in inheritance hierarchies, a CLib-state can be defined in terms of other CLib-states. For instance, the CLib-state `Be-Attached-To` is defined in terms of the CLib-state `Be-Touching`:

```
(Be-Attached-To has
    (superclasses   (Be-Touching)))
```

  In $\mathcal{ALM}$ however, only classes are organized into a hierarchy, but not functions. This issue is addressed by introducing state constraints stating that a domain property must hold in every state of the transition diagram in which the properties specializing it also hold. Let us take the example of the CLib-state `Be-Touching`, with two required attributes, and its subclass `Be-Attached`. We model this relation by the state constraint:
  $touching(X, Y)$ **if** $attached\_to(X, Y)$.

- Certain features of CLib have no translation into concepts of $\mathcal{ALM}$. One such example are defeasible laws, for instance soft preconditions for the execution of an action, which are not always necessary to hold in order for the occurrence of the action to be possible.

Regarding our second concern, the primary challenge was that the main reusable unit of $\mathcal{ALM}$ is *not* that of a class, as in CLib, but rather that of a *module*, a concept that does not exist in KM. This results from our knowledge representation thought pattern, which revolves around the task of specifying dynamic systems, not action classes in isolation. To address this issue, we grouped action classes of CLib by themes. Some of the themes we identified were: motion, accessibility, contact and attachment, resource management, collections, communication, or change in possession. The action classes corresponding to one theme were all placed in the same module, unless there were too many such classes. For instance, the theme of motion had to be broken down into several modules ($fundamental\_motion$, $locomotion$, $container\_motion$, etc.) to ensure their manageable size and readability. All these motion modules, however, formed part of the same theory stored in the library. Once action classes were assigned to a module, the $\mathcal{ALM}$ module was completed with the translation of axioms and of the related CLib components mentioned in these axioms (e.g., other classes of objects, CLib-states, other actions inhibiting the execution of the selected actions). Classes of CLib that were recurrently referenced in other classes were grouped in a high-level module upon which all other modules depended. Two examples are the hierarchy of entities, captured by module $entity\_ontology$, and the general definition of events and actions, included in a module called $general\_events$. Both of these modules form part of a theory $fundamental\_concepts$ stored in the library. Below, we show parts of the contents of these modules, after which we will explain some key points.

theory $fundamental\_concepts$
module $entity\_ontology$
class declarations
$entity :: universe$
$spatial\_entity :: entity$ ...

```
module general_events
  depends on entity_ontology
  class declarations
    event :: universe
      attributes
        agent : entity
        object : entity
        base : universe
        origin : spatial_entity
        destination : spatial_entity    ...
    actions :: event
    sequences_of_actions :: event
    role :: universe
  function declarations
    statics
      plays : entity × role × event → booleans
    constants
      target, medium, signal, resource : role
```

Notice first that the knowledge representation methodology of CLIB differs from that of $\mathcal{ALM}$ illustrated by the *commonsense_motion* theory shown earlier. In CLIB, all possible attributes of events are specified in the topmost class of the hierarchy of events, whereas in the $\mathcal{ALM}$ theory of *commonsense_motion*, we only introduced attributes in the subclasses of the pre-defined class *actions* as these attributes became relevant to an action class. Moreover, the attributes of the CLIB class *event* are all supposed to have the same meaning across subclasses of this class with the exception of *base*; in *commonsense_motion* attributes have names closely related to the meaning of the action class to which they belong (e.g., *grasper* and *grasped_thing* in an action class *grasp*). Which methodology is the best is still up to debate. Second, notice how object constants (i.e., functions of arity 0 such as *target*, *medium*, etc.) can be specified in the function declaration section of an $\mathcal{ALM}$ module, in a subsection preceded by the keyword **constants**.

A CLIB-inspired *fundamental_motion* Module

We further discuss our methodology of translating CLIB into $\mathcal{ALM}$ by briefly focusing on the *fundamental_motion* module that includes actions *move* and *carry*.[4]

When creating the *fundamental_motion* module, we took advantage of the features of $\mathcal{ALM}$, which encourage representing knowledge in as general terms as possible, and we introduced new classes to create a more general representation. For instance, CLIB contains a class of `Spatial-Relations`; we further divided this class into the classes *sym_sp_rel* and *antisym_sp_rel*, denoting symmetric and antisymmetric spatial relations respectively, to minimize the number of axioms in the encoding. We then declared the different spatial relations defined in CLIB as object constants of the $\mathcal{ALM}$ module. We introduced a new basic fluent absent in CLIB, *spatial_sit*, to describe the spatial situation between two spatial entities, and a new defined fluent, *known_spatial_sit*, to describe the spatial situation of a spatial entity in terms of a *place*.

[4]See http://tinyurl.com/m5a9zn3. The CLIB counterpart can be seen at http://tinyurl.com/k6z6ues.

In CLIB, the *agent* of a *carry* action is required (and similarly for the *object* of *move*). We specified this in our module via state constraints of the type

$$false \textbf{ if } \neg dom_{agent}(X), instance(X, carry).$$

Function dom_{agent} above is considered implicitly declared in module *general_events* where attribute *agent* is declared explicitly; $\neg dom_{agent}(X)$ is to be read as "X is not in the domain of function *agent*," meaning that "function *agent* is not defined on object X."

It is important to remark that our *fundamental_motion* $\mathcal{ALM}$ module contains a much smaller number of axioms and lines of code than its counterpart classes in CLIB – the relevant part of the declaration of action classes `Move` and `Carry` in CLIB totals approximately 300 lines, while our module contains around 90 lines. As a consequence, we believe that the $\mathcal{ALM}$ representation gains in readability. All this is largely because of (1) the way action effects and preconditions are specified in $\mathcal{ALM}$; (2) the introduction of fluent *spatial_sit*; and (3) the introduction of classes *sym_sp_rel* and *antisym_sp_rel* with their constant objects. We also noticed that the use of the CLIB knowledge representation methodology in which all actions *may have* an *agent* (or other attributes) forced us to introduced additional executability conditions for *move* in order to cover the case in which the optional attribute *agent* is actually specified. This indicates one disadvantage for the CLIB approach of declaring all possible attributes in the *event* class.

Conclusions and Future Work

In this paper, we have described a modular action language, $\mathcal{ALM}$, suitable for the specification of large dynamic systems and for the development of knowledge representation libraries. We have presented our approach of building a large $\mathcal{ALM}$ library by translating CLIB—a library of general components selected from lexical and ontological resources based on a thorough methodology.

This work enables an informal comparison between two knowledge representation languages: $\mathcal{ALM}$ and KM, the language of CLIB. $\mathcal{ALM}$ has the advantage of a concise representation of effects and preconditions of actions, as well as an increased readability due to the structuring in the same module of knowledge on a common theme. On the other hand, KM allows the description of *classes* of *compound* actions, which were not addressed by the current translation. For instance, an action like `Disperse`–several objects leaving the same place— cannot be elegantly represented in $\mathcal{ALM}$ currently, although *instances* of compound actions can be represented. Future work on $\mathcal{ALM}$ will address this issue.

A strength of CLIB is the ease of finding relevant components in it, as the library provides users with well-documented specifications of its components. This aspect was not yet explored in the context of $\mathcal{ALM}$, but will have to be studied in the future.

Acknowledgments. This research was funded by SRI International and Vulcan Inc.

References

Balduccini, M., and Gelfond, M. 2003. Diagnostic Reasoning with A-Prolog. *TPLP* 3(4–5):425–461.

Balduccini, M. 2012. *A "Conservative" Approach to Extending Answer Set Programming with Non-Herbrand Functions*. Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz. Springer Verlag, Berlin. 23–39.

Baral, C., and Gelfond, M. 2000. *Reasoning Agents in Dynamic Domains*. Norwell, MA: Kluwer Academic Publishers. 257–279.

Barker, K.; Porter, B.; and Clark, P. 2001. A Library of Generic Concepts for Composing Knowledge Bases. In *Proceedings of the 1st international conference on Knowledge capture*, K-CAP '01, 14–21. New York, NY, USA: ACM.

Chaudhri, V. K.; John, B. E.; Mishra, S.; Pacheco, J.; Porter, B.; and Spaulding, A. 2007. Enabling experts to build knowledge bases from science textbooks. In *Proceedings of the 4th international conference on Knowledge capture*, K-CAP '07, 159–166. New York, NY, USA: ACM.

Chaudhri, V. K.; Clark, P. E.; Mishra, S.; Pacheco, J.; Spaulding, A.; and Tien, J. 2009. Aura: Capturing knowledge and answering questions on science textbooks. Technical report, SRI International.

Chaudhri, V.; Dinesh, N.; and Inclezan, D. 2014. Three Lessons in Creating a Knowledge Base to Enable Explanation, Reasoning and Dialog. *Advances in Cognitive Systems* 3:183–200.

Clark, P. E., and Porter, B. 2004. KM – The Knowledge Machine 2.0: Users Manual. Retrieved from the web page: http://www.cs.utexas.edu/users/mfkb/km/userman.pdf.

Clark, P.; Chaw, S.; Barker, K.; Chaudhri, V.; Harrison, P.; John, B.; Porter, B.; Spaulding, A.; Thompson, J.; and Yeh, P. Z. 2007. Capturing and answering questions posed to a knowledge-based system. In *Proceedings of Fourth International Conference on Knowledge Capture*.

Erdoğan, S., and Lifschitz, V. 2006. Actions as special cases. In *Principles of Knowledge Representation and Reasoning: Proceedings of the International Conference*. 377–387.

Gelfond, M., and Inclezan, D. 2009. Yet Another Modular Action Language. In *Proceedings of SEA-09*, 64–78. University of Bath Opus: Online Publications Store.

Gelfond, M., and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the International Conference on Logic Programming (ICLP'1988)*, 1070–1080.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9(3/4):365–386.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on AI* 3(16):193–210.

Giunchiglia, E., and Lifschitz, V. 1998. An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 623–630. AAAI Press.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic Causal Theories. *Artificial Intelligence* 153(1–2):105–140.

Gunning, D.; Chaudhri, V. K.; Clark, P.; Barker, K.; Chaw, S.-Y.; Greaves, M.; Grosof, B.; Leung, A.; McDonald, D.; Mishra, S.; Pacheco, J.; Porter, B.; Spaulding, A.; Tecuci, D.; and Tien, J. 2010. Project Halo–Progress Toward Digital Aristotle. *AI Magazine* 31(3):33–58.

Inclezan, D., and Gelfond, M. 2011. Representing Biological Processes in Modular Action Language ALM. In *Proceedings of the 2011 AAAI Spring Symposium on Formalizing Commonsense*, 49–55. AAAI Press.

Lifschitz, V., and Ren, W. 2006. A Modular Action Description Language. Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI), 853–859.

McCain, N., and Turner, H. 1997. Causal Theories of Action and Change. In *Proceedings of AAAI-97*, 460–465.

McCarthy, J., and Hayes, P. J. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. Edinburgh University Press. 463–502.

Turner, H. 1997. Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *Journal of Logic Programming* 31(1–3):245–298.