TLP **16** (5–6): 800–816, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000363

CoreALMLib: *An $\mathcal{ALM}$ library translated from the* Component Library

DANIELA INCLEZAN
Miami University, Oxford OH, 45056, USA
(*e-mail:* inclezd@miamioh.edu)

submitted 6 August 2016; revised 8 July 2016; accepted 22 August 2016

Abstract

This paper presents CoreALMLib, an $\mathcal{ALM}$ library of commonsense knowledge about dynamic domains. The library was obtained by translating part of the Component Library (CLib) into the modular action language $\mathcal{ALM}$. CLib consists of general reusable and composable commonsense concepts, selected based on a thorough study of ontological and lexical resources. Our translation targets CLib *states* (i.e., *fluents*) and *actions*. The resulting $\mathcal{ALM}$ library contains the descriptions of 123 action classes grouped into 43 reusable modules that are organized into a hierarchy. It is made available online and of interest to researchers in the action language, answer-set programming, and natural language understanding communities. We believe that our translation has two main advantages over its CLib counterpart: (i) it specifies axioms about actions in a more elaboration tolerant and readable way, and (ii) it can be seamlessly integrated with ASP reasoning algorithms (e.g., for planning and postdiction). In contrast, axioms are described in CLib using STRIPS-like operators, and CLib's inference engine cannot handle planning nor postdiction.

KEYWORDS: action language, commonsense library, knowledge reuse, dynamic domains

1 Introduction

In the field of knowledge representation about dynamic domains, an important advancement in the last decade was the design of so-called *modular* action languages, which focus on the reuse of knowledge and creation of libraries in general, and the representation of actions in terms of other actions in particular. $\mathcal{ALM}$ (Action Language with Modules) (Inclezan and Gelfond 2016) was introduced to address these issues. The capabilities of the language were tested in the context of Project Halo (Gunning *et al.* 2010): specialized knowledge about a biological process was encoded in $\mathcal{ALM}$ and then used to answer non-trivial questions (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 $\mathcal{ALM}$

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

with reasoning algorithms written in Answer Set Prolog (ASP) (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991). However, we were not sure what criteria to use for the selection of further concepts, general enough to deserve inclusion in our library.

An answer to this 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 minimal 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 *et al.* 2014). CLIB (Barker *et al.* 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 part of the library, in its revised form resulting from our analysis, into $\mathcal{ALM}$. The resulting $\mathcal{ALM}$ library combines the advantages of CLIB with established knowledge representation methodologies developed in the *action language* community: concise and elaboration tolerant representations of action effects and preconditions, and an easy coupling with reasoning algorithms encoded in ASP (Gelfond and Lifschitz 1998). In the future, the experience of creating a CLIB-inspired library will allow us to further explore how $\mathcal{ALM}$ libraries should be structured, queried, used, and made available to the public.

Related work. A previous version of this work was presented in an earlier paper (Inclezan 2015) whose goal was to introduce $\mathcal{ALM}$ and explain how enforcing the thought pattern of our language on a translation from CLIB results in much more concise representations. Instead, in the current paper we describe in more detail a translation that is closer to CLIB and can potentially be automated. We also make the library available online and discuss how it can be used.

In addition to CLIB, several linguistic resources, including VerbNet (Kipper-Schuler 2005), WordNet (Miller *et al.* 1990) and FrameNet (Baker *et al.* 1998), contain information about verbs, sometimes organized into an inheritance hierarchy or into sets of synonyms. Verbs may sometimes be accompanied by axioms, but these are expressed in informal terms, and therefore cannot be readily used to create an $\mathcal{ALM}$ library of commonsense knowledge. A modular action language with similar goals to $\mathcal{ALM}$'s is MAD (Lifschitz and Ren 2006; Erdoğan and Lifschitz 2006). There is a collection of MAD libraries (Erdoğan 2008), but its fifteen library modules only describe eight action classes (compared to 146 in CLIB), and are less expressive as a result.

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

2 Modular action language $\mathcal{ALM}$

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 organized into a hierarchy) and a *structure* (i.e., an interpretation of some of the symbols in the theory). A *module* is a collection of declarations of sorts (i.e., 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 and can be organized into a hierarchy (a DAG). If a module M_1 contains the statement "**depends on** M", then the declarations and axioms of module M are implicitly part of M_1.

We briefly illustrate the syntax of $\mathcal{ALM}$ via some examples. For a formal description of the language, we direct the reader to the paper that introduces $\mathcal{ALM}$ (Inclezan and Gelfond 2016). Boldface symbols denote keywords of the language; identifiers starting with a lowercase letter denote constant symbols; and identifiers starting with an uppercase letter denote variables.

Sorts are organized into a hierarchy with root **universe** and pre-defined sorts **actions** and **booleans**. The sort hierarchy is specified via the specialization construct "::". For instance, we say that *points* and *things* are subsorts of **universe** and *agents* is a subsort of *things* by:

points, *things* :: **universe**
agents :: *things*

We use the same construct to define action classes, if needed as special cases of other action classes. Declarations of action classes include the specification of *attributes*, which are intrinsic properties. For instance, the statements:

move :: **actions**
  attributes
    actor : *agents*
    origin, *dest* : *points*

define *move* as having three attributes, *actor*, *origin*, and *dest* – (possibly partial) functions mapping elements of *move* into elements of *agents*, *points*, and *points* respectively.

Properties of objects of a dynamic system are represented using functions. Functions are partitioned in $\mathcal{ALM}$ into *fluents* (those that can be changed by actions) and *statics* (those that cannot); each of these are further divided into *basic* and *defined*, where defined functions can be viewed just as means to facilitate knowledge encoding. Basic fluents are subject to the law of inertia. In our example, the location of *things* is encoded by the

fluent basic *loc_in* : *things* → *points*.

Axioms that can appear in a module are of four types – dynamic causal laws, state constraints, definitions for defined functions, and executability conditions. The axiom:

$occurs(X)$ **causes** $loc_in(A) = D$ **if** $instance(X, move)$, $actor(X) = A$, $dest(X) = D$.

is a dynamic causal law saying that the actor of a *move* action will be located at the destination after the execution of the action.

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

The second part of a system description is its *structure*, which represents information about a specific domain: instances of sorts (including actions) and values of statics. For example, a domain that is about John and Bob, and their movements between two points, *a* and *b*, may be described as follows:

$john, bob$ **in** *agents*
a, b **in** *points*
$go(X, Y)$ **in** *move*
  $actor = X$
  $dest = Y$

Action $go(X, Y)$ is an instance schema that stands for all actions of this form obtained by replacing X and Y with instances of *agents* and *points*, respectively. One such action is $go(john, a)$ for which attributes have the values $actor(go(john, a)) = john$ and $dest(go(john, a)) = a$.

The semantics of ALM is given 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 2013), an extension of ASP by non-Herbrand functions. The states and transitions of the corresponding transition diagram are 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 above is encoded as:

$$loc_in(A, I+1) = D \leftarrow instance(X, move),\ occurs(X, I),\ actor(X) = A,\ dest(X) = D$$

followed by replacing variables (other than I) by constants of the appropriate sorts. The structure is encoded using statements like:

$is_a(john, agents).$
$is_a(go(john, a), move).$ $\quad actor(go(john, a)) = john.$ $\quad dest(go(john, a)) = a.$ $\quad (etc.)$

3 CLib and KM

CLib (Barker *et al.* 2001) is a library of general, reusable, composable, and interrelated components of knowledge. Notions included in CLib were selected using a solid methodology relying on linguistic and ontological resources such as WordNet (wordnet.princeton.edu), FrameNet (framenet.icsi.berkeley.edu), VerbNet (verbs.colorado.edu/verb-index), 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. These are criteria that we want for a core ALM library as well.

The CLib 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 (FOL) semantics. The language uses its own syntax, which is mainly syntactic sugar for FOL (Clark and Porter 2004). KM distinguishes between two basic concepts: *class* and *instance*.

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

(a) The Declaration of the Action Class Obstruct

```
(Obstruct has
  (superclasses (Make-Inaccessible))
  (wn20-synset ((:set
    (:triple "obstruct" 2 "v") (:triple "block" 10 "v") (:triple "close_up" 1 "v")
    (:triple "jam" 7 "v")      (:triple "impede" 2 "v") (:triple "occlude" 1 "v")))))
(every Obstruct has
  (object ((a Tangible-Entity)))
  (resulting-state ((a Be-Obstructed)))
  (add-list (
    (:triple (the resulting-state of Self) object (the object of Self))
    (if (has-value (the agent of Self))
     then (forall (the agent of Self)
      (:triple It agent-of (the resulting-state of Self))))))
  (preparatory-event (((:default
    (if (has-value (the agent of Self))
     then (a Move with
        (object     ((the agent of Self)))
        (destination ((a Spatial-Entity with (is-at ((the object of Self))))))))))))
```

(b) The Declaration of the State Class Be-Obstructed

```
(Be-Obstructed has
  (superclasses   (Be-Inaccessible))
  (wn20-synset ((:set (:triple "obstructed" 1 "a")))))
(every Be-Obstructed has
  (object ((a Entity))))
```

(c) The Definition of the Slot Instance is-at

```
(is-at has
  (instance-of (Spatial-Relation))
  (domain (Spatial-Entity))
  (range (Spatial-Entity))
  (cardinality (N-to-N))
  (fluent-status (*Inertial-Fluent)))
```

(d) The Definition of the Slot Instance object

```
(object has
  (instance-of (Participant-Relation))
  (domain (Event))
  (range (Entity))
  (cardinality (N-to-N))
  (fluent-status (*Inertial-Fluent)))
```

Fig. 1. Examples of declarations/Definitions of CLib components.

In CLib, there are three main classes – Entity, Event, and Role – and the additional built-in class Slot. All are subclasses of the root class Thing. Events are divided into Actions, Activities, and States (which should not be confused with states of the transition diagram). An example of a CLib action is Obstruct, whose declaration can be seen in Figure 1(a). It starts with the specification of properties of the class itself: it is a subclass of the action class Make-Inaccessible and it has an associated set of WordNet 2.0 synonyms that includes the verbs "obstruct" and "jam." In the second part of the declaration, properties of *instances* of the action class are specified. It is asserted that each instance of Obstruct has an object that must be an instance of Tangible-Entity. In practice, this results in the automatic creation of a Skolem constant named _Tangible-Entity*N* to denote the object of an Obstruct instance, where *N* is a number. Next, resulting-state and add-list together specify the effect of the execution of an Obstruct action: the object will be obstructed (by the agent, if it was defined). The default preparatory-event says

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

that, if the `Obstruct` action has an agent, then the agent should be where the object is, in order for the action execution to be possible. Note that failure to meet default axioms only generates warnings and does not prevent actions from actually being executed. The declaration of `Obstruct` also contains the description of a test case, information for generating English text, and a WordNet 1.6 synset, which were not included in the figure.

Expressions like `superclasses` and `object` in the declaration of `Obstruct` are *slots* (i.e. instances of the class `Slot`): they denote binary relations that hold between the class or instances of the class and the value denoted by the expressions that follow. Two of the slots, `superclasses` and `add-list`, are built-ins of KM, while the other ones are specific to CLib. The definitions of slots `is-at` and `object` can be seen in Figure 1(c) and (d) respectively.

States of CLib correspond to fluents in action language terminology, but they are represented by classes, which means that they can be organized into an inheritance hierarchy. An example of a CLib state is `Be-Obstructed` from Figure 1(b) that is a subclass of the state `Be-Inaccessible` and corresponds to the first sense of the adjective "obstructed" in WordNet 2.0. Every instance of `Be-Obstructed` is asserted to have an `object` that must be an `Entity`.

The CLib library was integrated in two systems developed at SRI International, SHAKEN and its successor AURA (Chaudhri *et al.* 2007; Clark *et al.* 2007), and was extensively tested as a result. In AURA, domain experts were able to encode new knowledge by building upon the general concepts of CLib in a speedy manner and with no or minimal involvement from knowledge engineers. Their encodings were used to answer questions from an Advanced Placement test suite. At least 70% of these questions 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.

4 A (partial) translation from KM into $\mathcal{ALM}$

In this work, we only intend to translate part of CLib into $\mathcal{ALM}$ and refer to this portion as *a*CLib. Specifically, *a*CLib focuses on actions and states, but ignores roles, activities, actions with subevents or duration, and text generation information associated with CLib components, which either deserve more study (roles, actions with duration) or exceed the scope of this project. Thus, *a*CLib contains all but 23 of the 146 actions of CLib and all but four of its 33 states. The *a*CLib concepts were manually translated into $\mathcal{ALM}$. The translation presented here covers the fragment of KM used in *a*CLib. To give a simple example, in KM, one may assert that a slot of a specific instance maps into a given number of other instances by using the expression "`exactly` *n*"; however in all occurrences of this expression in *a*CLib, *n* only ranges over {0, 1, 2}.

Two main differences between KM and $\mathcal{ALM}$ posed problems to the translation: (1) the unique name assumption is part of the semantics of $\mathcal{ALM}$, but not of KM and (2) FOL quantifiers do not exist in $\mathcal{ALM}$, but they do in KM. The existential quantifier in KM, "`(a <c>)`", is constructive and introduces a Skolem

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

constant, as explained in Section 3. KM implements a complex unification algorithm for automatically introduced Skolem constants. In $\mathcal{ALM}$, the user of the library introduces and names instances when specifying the structure of a system description. We translated the existential quantifier using constraints to produce a similar effect to KM.

An additional difficulty stems from the fact that CLib states (i.e., fluents) are represented as classes and thus can be organized into a hierarchy and have an arbitrary number of parameters. Instead, fluents of $\mathcal{ALM}$ are represented as functions and have a fixed arity. To address these problems, we generally translated one CLib state using two $\mathcal{ALM}$ fluents with different arities and added state constraints to encode the inheritance relation between CLib states.

In what follows, we briefly describe the translation from KM into $\mathcal{ALM}$. We start with definitions of slots, then move on to declarations of states and actions. We ignore the difference between the capitalization conventions of KM versus $\mathcal{ALM}$ for conciseness and clarity. In KM code, we put placeholders for constants inside chevrons (<>); in $\mathcal{ALM}$ code, we omit the chevrons.

4.1 Slots

Consider a slot s defined as follows:

```
(<s> has (instance-of (<c>))
         (domain (<c1>))
         (range (<c2>))
         (cardinality (<card>))
         (fluent-status (<status>)))
```

If card is N-to-N, then it is translated as the $\mathcal{ALM}$ function

s : c1 × c2 → **booleans**

and if card is N-to-1 as

s : c1 → c2.

There are no slots with cardinality 1-to-N in *a*CLib. If status is *Non-Fluent, then s is a basic static in $\mathcal{ALM}$. Otherwise, it is a basic (i.e., inertial) fluent, unless:

(1) c is Participant-Relation or

(2) c is Spatial-Relation and c1 is Event.

As an example, the slot is-at from Figure 1(c) is encoded as the basic fluent

is_at : *spatial_entity* × *spatial_entity* → **booleans**.

If conditions (1) or (2) above are satisfied, then s describes an intrinsic property of an event (and thus of an action too) and is translated as an attribute of the pre-defined $\mathcal{ALM}$ class **actions**.

4.2 States

CLib states encode fluents as classes. They do not have a fixed arity, rather their parameters are represented via slots that are instances of the subclass Participant-Relation of Slot (e.g., object, instrument). Certain participant relations are *required* for a state: if the declaration of a state c says that

```
(every <c> has (<r> ((a <c1>))))
```

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

then r is required for c. We analyzed participant relations w.r.t. states and determined that (i) object was required in all states; (ii) a few states had a secondary required participant relation (e.g., base in Be-Confined referring to the place where the object is confined); and (iii) a few other participant relations were only sometimes associated with some of the states (e.g., instrument in Be-Blocked denoting the thing with which the object is blocked). As a result, if a state class Be-<f> only had one required participant relation, object, we translated it by creating a basic fluent

is_f : c1 → **booleans**

where c1 is the most specific class membership information for the participant relation object with respect to the state Be-<f>. For instance, for Be-Obstructed, we created the basic fluent

is_obstructed : *spatial_entity* → **booleans**

If the state had a second (required/ associated) participant relation r with range c2, we created (instead/ in addition, respectively) the basic fluent

f_prep : c1 × c2 → **booleans**

where prep is the preposition that is normally associated in English with the participant relation r (e.g., for instrument, it would be the preposition "with"). As an example, the Be-Blocked state would be translated by two fluents:

is_blocked : *spatial_entity* → **booleans**

blocked_with : *spatial_entity* × *entity* → **booleans**

Whenever we translated a CLIB state by introducing two ALM fluents with different arities, we connected the two via axioms of the style:

is_blocked(*O*) **if** *blocked_with*(*O*, *I*).

¬*blocked_with*(*O*, *I*) **if** ¬*is_blocked*(*O*).

We expressed the subclass relationship between CLIB states by introducing state constraints. For example, we said that Be-Obstructed is a subclass of Be-Inaccessible via the axiom:

¬*is_accessible*(*O*) **if** *is_obstructed*(*O*).

4.3 Actions

We replaced the CLIB class Action with the predefined ALM class **actions**. We translated the information about the superclasses of an action class of *a*CLIB

```
(<c> has (superclasses (<s1> ... <sn>))
```

using the specialization construct (i.e., "::") of ALM:

c :: s1, ..., sn.

The information about WordNet 2.0 synsets was integrated in the online tool (see Section 5).

The description of properties of every instance of the class can be divided into two parts that we address in separate subsections below: (1) *attribute declarations* – the description of the values of participant or spatial relations like object, agent, origin, destination, etc. that describe intrinsic properties of the action; and (2) *axioms* – the specification of the values of slots resulting-state, defeats,

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

`add-list`, `del-list`, `pcs-list`, etc. that describe the effects and preconditions for the execution of the action.

4.3.1 Attribute Declarations

In what follows, by `<attr>` we denote a slot that stands for a participant or spatial relation describing an inherent property of instances of the action class and refer to it as an *attribute*. Consider the following part of the declaration of action class `c`:

```
(every <c> has (<attr> ((a <c1>))))
```

In KM, this generates a Skolem constant for each instance of `c`, which cannot be done in $\mathcal{ALM}$. We encode the statement via an attribute declaration (when needed) and several axioms, one of them requiring the introduction of a defined fluent, as explained below. If `attr` is not an attribute of any superclass of `c`, we add to the declaration of `c` the attribute:

`attr` : `c1` → **booleans**

If `attr` is an attribute of a superclass `s` of `c`, translated in $\mathcal{ALM}$ as `attr` : `c2` → **booleans** such that `c2` is different from `c1` (possibly a superclass of `c1`), we add the constraint

$false$ **if** $instance(X, \texttt{c})$, $\texttt{attr}(X, A)$, $\neg instance(A, \texttt{c1})$.

In both cases, we add a defined static

$defined_\texttt{attr}$: `c` → **booleans**

that is true when there is a value for the attribute and require it to be true for all instances of `c`:

$defined_\texttt{attr}(X)$ **if** $\texttt{attr}(X)$.

$false$ **if** $instance(X, \texttt{c})$, $\neg defined_\texttt{attr}(X)$.

Constraints on the values of attributes in *a*CLIB are encoded using state constraints of $\mathcal{ALM}$, sometimes preceded by the declaration of new defined statics that are needed to deal with the lack of quantifiers in $\mathcal{ALM}$. The constraint that `attr` must range over instances of `c1`

```
(every <c> has (<attr> ((must-be-a <c1>))))
```

is translated as

$false$ **if** $instance(X, \texttt{c})$, $\texttt{attr}(X, A)$, $\neg instance(A, \texttt{c1})$.

Similarly if the keyword `mustnt-be-a` is used instead. The constraint that `attr` must map into at most one instance of `c1` for instances of `c` (after unification):

```
(every <c> has (<attr> ((at-most 1 <c1>))))
```

is captured by the state constraint

$\neg\texttt{attr}(X, A1)$ **if** $\texttt{attr}(X, A2)$, $A1 \neq A2$, $instance(X, \texttt{c})$, $instance(A1, \texttt{c1})$, $instance(A2, \texttt{c1})$.

Similarly for `at-most 2`, just with a more complex axiom. If `at-most` is substituted by `at-least`, we have axioms similar to the ones for `(a <c1>)` in $\mathcal{ALM}$. For `at-least 2`, we would first introduce a defined fluent $at_least_2_\texttt{attr}$: `c` → **booleans**

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

that is true whenever there are at least two distinct values of class `c1` for `attr`; then we would add the state constraint

false **if** *instance*(X, `c`), ¬*at_least_2_attr*(X).

The keyword `exactly` *n* in a constraint would be translated by putting together the translations of `at-least` *n* and `at-most` *n* if *n* is 1 or 2. If *n* is 0, we would add the state constraint

false **if** *instance*(X, `c`), `attr`(X, A), *instance*(X, `c1`).

Other constraints require the value of an attribute to be the same, different, or unifiable ("&") with that of some other attribute or expression. For the constraints

(a) `(every <c> has (<attr1> ((the <attr2> of Self))))`

(b) `(every <c> has (<attr1> ((excluded-values (the <attr2> of Self))))`

(c) `(every <c> has (<attr1> ((constraint (TheValue & (the <attr2> of Self))))))`

the corresponding state constraints in the ALM translation are

(a) `attr1`(X, V) **if** `attr2`(X, V), *instance*(X, `c`).

(b) *false* **if** *instance*(X, `c`), `attr1`(X, A), `attr2`(X, A).

(c) In ALM, we must say that the values are the same (via a defined static and several axioms).

4.3.2 Axioms

We present the translation of the most common types of axioms encountered in *a*CLib. More complex axioms were translated in a similar way.

Action Preconditions were specified using lists of properties that must be true (`pcs-list`) or false (`ncs-list`) for the action to be executable. Consider the axiom:

```
(every <c> has (pcs-list ((forall (the <attr> of Self)
                            (:triple It object-of (a Be-<f>))))))
```

where `object-of` is the inverse of `object`. It says that, in order for an action of class `c` to be executable, its attribute `attr` must be in a `Be-<f>` state. Thus, we translate it as:

impossible *occurs*(X) **if** *instance*(X, `c`), `attr`(X, A), ¬*is_f*(A).

Another common type of axiom was the one below:

```
(every <c> has
  (pcs-list
    ((forall (the <attr1> of Self)
      (:triple It object-of (a Be-<f> with (<attr2> ((the <attr2> of Self)))))))))
```

This says that the action's attribute `<attr1>` must be in a `Be-<f>` state that has attribute `attr2` mapped into the action's same attribute. If we denote by `prep` the preposition associated with the attribute `attr2` in the context of state `Be-<f>`, then the ALM translation would look as follows:

impossible *occurs*(X) **if** *instance*(X, `c`), `attr1`(X, $A1$), `attr2`(X, $A2$), ¬ `f_prep`($A1$, $A2$).

Negative preconditions (`ncs-lists`) are translated in a similar way.

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

Action Effects were normally encoded by a `resulting-state` referenced in an `add-list` (properties that will hold after the action execution), or a `defeats` state referenced in a `del-list` (properties that will not hold). The simplest form of an `add-list` of *a*CLIB can be seen below:

```
(every <c> has
  (resulting-state ((a Be-<f>)))
  (add-list
   ((:triple (the resulting-state of Self) object (the <attr> of Self)))))
```

saying that as a result of the execution of an action of class `c`, its attribute `attr` will now be the object of a `Be-<f>` state. Its translation into $\mathcal{ALM}$ is:

occurs(X) **causes** *is_*`f`(A) **if** *instance*$(X,$ `c`$)$, `attr`(X, A).

A more complex form, containing an `if – then` expression is:

```
(every <c> has
  (resulting-state ((a Be-<f>)))
  (add-list (
  (if (has-value (the <attr2> of Self))
     then (:triple (the resulting-state of Self) <attr2> (the <attr2> of Self))
     else (:triple (the resulting-state of Self) object (the <attr1> of Self))))))
```

In addition to what was mentioned for the previous axiom, this says that, if there is value for the action's attribute `attr2`, then it will also be the `attr2` of the `Be-<f>` state. This requires using the binary version of the corresponding fluent in $\mathcal{ALM}$. Assuming that `prep` is the preposition associated with attribute `attr2` in the binary version of `f`, then the $\mathcal{ALM}$ translation would be:

occurs(X) **causes** `f_prep`$(A1, A2)$ **if** *instance*$(X,$ `c`$)$, `attr1`$(X, A1)$, `attr2`$(X, A2)$.

occurs(X) **causes** *is_*`f`(A) **if** *instance*$(X,$ `c`$)$, `attr1`(X, A), $\neg$*defined_*`attr2`(X).

Finally, let us consider an example of a `del-list`:

```
(every <c> has
  (defeats ((allof (the object-of of (the <attr> of Self))
       where ((the classes of It) = Be-<f>))))
  (del-list ((forall (the defeats of Self)
                (:triple (It) object (the <attr> of Self) )))))
```

This says that the action's attributes `attr` that are in a `Be-<f>` state should no longer be in this state after the execution of the action. We translate it in $\mathcal{ALM}$ as:

occurs(X) **causes** $\neg$*is_*`f`(A) **if** *instance*$(X,$ `c`$)$, `attr1`(X, A), *is_*`f`(A).

Defeasible axioms. Some action classes of *a*CLIB also contained a set of axioms that were defaults (they did not prevent the action from being executed, just produced warnings). They were specified as `soft-pcs-list` or `preparatory-events`. We translated both in a similar way to `pcs-list`, but marked them as optional when including their translation in the library, as further discussed in Section 5. This allows the user to decide whether they apply to a particular domain to be represented and should be included in the system description or not.

4.4 Remarks

In addition to the direct translation described above, we sometimes added extra axioms to the $\mathcal{ALM}$ translation for consistency or in order to account for

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

specifications that we felt were missing. An example of the first is when one action class contained an executability condition but the action class with an opposite effect did not contain the counterpart (e.g., `Unblock` and `Block`). For `Move`, we added the restrictions that the origin and destination should have at most one value, to distinguish it from action classes that would otherwise be more suitable (e.g., `Move-Together` or `Move-Apart`). Additionally, since we focused on discrete actions that occur instantaneously, we changed the names of a couple of actions to make them sound less as processes and more like discrete actions (e.g., `Hold` was renamed as *take_hold*).

Axioms in the ALM translation are more succinct and elaboration tolerant that the STRIPS-like add and delete lists used in CLIB. On the other hand, constraints on attributes are expressed more concisely in KM using FOL quantifiers; in ALM axioms and extra statics are needed. This can be easily addressed in ALM by adding extra keywords and expanding the language with aggregates in the spirit of Gelfond and Zhang (2014), which we plan to do in the near future.

Formulating and proving a formal result on the soundness of our translation is non-trivial because KM is not exactly FOL and thus its semantics are not completely clear. This task deserves further investigation that will be the subject of another paper. We can say however that the translation is faithful to the intended meaning of *a*CLIB concepts.

5 The translated library COREALMLIB

5.1 Organizing the library into modules

The translation from KM (as used in *a*CLIB) into ALM was the first step in producing an ALM library of core commonsense knowledge. One additional step was needed however, because the basic concept of ALM, that of a *module*, is a higher-level one than the main concepts of KM (*class* and *instance*). An ALM module is a reusable piece of knowledge on a specific theme that groups together declarations of classes (including action classes) and functions, as well as axioms about these. CLIB has no concept similar to a module. Determining what modules to create, what functions and action classes of the translation to include in each module, and how to organize modules into a dependency hierarchy was not a trivial task.

We started by applying the guidelines for creating modular ALM representations provided by Inclezan and Gelfond (2016). Specifically, we started from the top of the *a*CLIB class hierarchy, and gradually built and tested modules capturing knowledge about actions, while reusing previously written modules as much as possible. This required us to create a root module that contained the translations of classes `Event` and `Action` from *a*CLIB and the general part of the `Entity` hierarchy. All possible participant and spatial relations of an action (i.e., *attributes* in ALM terminology) were also included in this module called *entity_event_and_action*. We identified fifteen major themes that allowed us to group the specific action classes into modules. For this purpose, we analyzed: (i) the hierarchies of state and action classes; (ii)

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

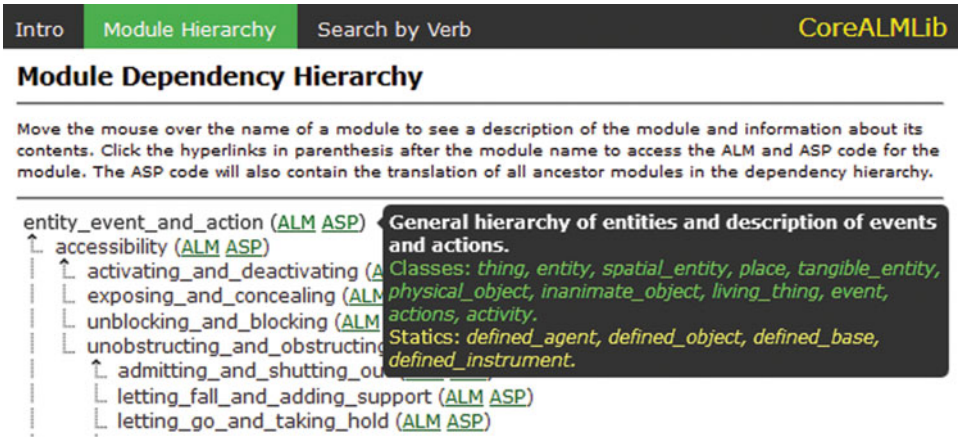


Fig. 2. Online tool: viewing the module dependency hierarchy.

the fluents relevant to each action class; and (iii) the list of neighboring concepts listed in the CLib user interface, which sometimes included an opposite action (e.g., `Unobstruct` for `Obstruct`). We placed in the same module opposite actions, actions that affected or had preconditions described in terms of the same CLib state, and sometimes subclasses of an action class. We strove to balance the size of a module with the depth of the (part of the) action class hierarchy that it captured and also considered the resulting depth of the module dependency hierarchy. Because of this, we sometimes placed subclasses of an action class in a separate module. Optional axioms (i.e., axioms resulting from the translation of default statements of *a*CLib) and any needed declarations were placed in a separate leaf module that depended on the module containing the declaration of the action class. The resulting library, CoreALMLib, consists of 43 modules. It encodes information about 123 action classes. Each module contains the description of one to six action classes. The depth of the module hierarchy is three (four if counting modules containing optional axioms), which we believe to be manageable.

5.2 Online tool

The CoreALMLib library is available online at the web page http://tinyurl.com/z6n9fmx. Users can download the entire library and a prototype translator from $\mathcal{ALM}$ into ASP from the main page. An additional page allows users to see the module dependency hierarchy and download individual modules (or their translation into ASP). When the user moves the mouse over the title of a module, a description of the module is displayed, containing the general purpose of the module, together with the list of action classes and functions declared in it (those that are new compared to ancestor modules in the hierarchy). An example can be seen in Figure 2.

A third page contains a lookup table for which the keys are English verbs or adjectives accompanied by WordNet sense numbers – see Figure 3. These words were extracted from the information about WordNet 2.0 synonym sets of *a*CLib action and state classes. Additionally, the table contains the definition of the word

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

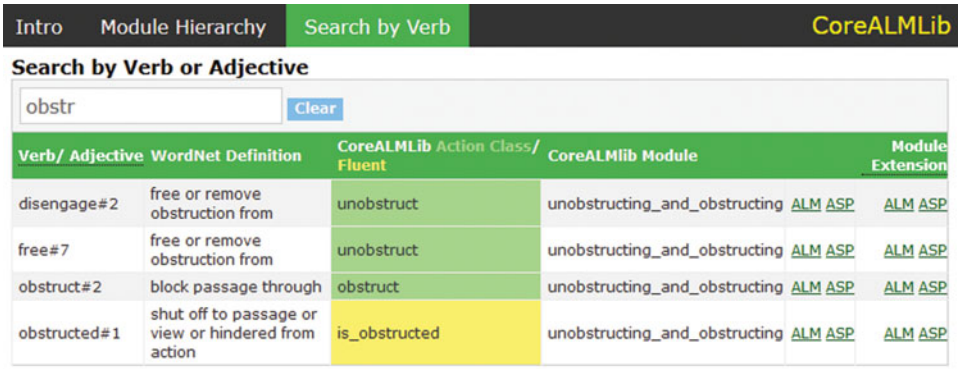


Fig. 3. Online tool: searching by english verb or adjective.

sense from WordNet, the name of the corresponding CoreALMlib action class or fluent and the module in which it can be found with links to the $\mathcal{ALM}$ and ASP code, and links to an extended module with optional axioms, if it exists. The table is searchable by verb/ adjective and WordNet definition (i.e., the two leftmost columns), as it can be seen in the example in Figure 3.

5.3 Testing and using CoreALMlib

We tested the library using test cases provided in CLib. Additionally, we encoded some scenarios that required the use of multiple modules for representation.

In order to create an $\mathcal{ALM}$ encoding of a particular domain, the user would first determine which CoreALMlib modules are relevant. For that, the "Search by Verb or Adjective" capability of the online tool should be used. Afterward, the user can either use $\mathcal{ALM}$ or ASP to expand the general knowledge from the library modules with knowledge particular to the domain. We explain here the first option and take as an example the following text:

"*The wrestler restrained his opponent.*"

By searching for the verb "restrain" in the online tool, the user would determine that the relevant CoreALMlib action class is *restrain* and that it can be found in the module *unrestraining_and_ restraining*. She would create a system description titled *wrestler_and_opponent* and import into its theory this library module:

system description *wrestler_and_opponent*
theory *wrestler_and_opponent*
import from *coreALMlib* **module** *unrestraining_and_restraining*

The import statement is equivalent to copying the contents of the imported module and of all its ancestors into the theory (e.g., *unobstructing_and_obstructing*). Next, the user would create a structure defining the entities mentioned in the sentence as instances of classes in the theory

structure *wrestler_and_opponent*
instances
wrestler, *opponent* **in** *living_entity*

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

and the event in the sentence as an instance of *restrain* in which the agent (the "doer") is the *wrestler* and the object (the one affected by the action) is the *opponent*.

r **in** *restrain*
  $agent(wrestler) = true$
  $object(opponent) = true$

(Note that even this simple domain cannot be encoded using the MAD libraries in Erdoğan's thesis (2008) in a comparably simple way.) The system description would be translated using the prototype translator from $\mathcal{ALM}$ into ASP. The user would have to add to the resulting logic program a predefined module for temporal projection and a history (Gelfond and Khal 2014). For our scenario, the history would specify that action r was observed to have happened at time step 0:

$hpd(r, 0)$

and that neither the wrestler nor the opponent were initially restrained:

$obs(is_restrained(wrestler), false, 0)$
$obs(is_restrained(opponent), false, 0)$.

Answer sets of the program will indicate that, as a result of action r happening at time step 0, the opponent will be restrained at the end of the story.

If we added the question "*What would need to happen in order for the opponent to be able to move freely?*" then the user would need to expand the theory of the initial system description by at least importing the CoreALMLib module *motion* that contains the description of action *move* mentioned in the question. Note that this module specifies that the action cannot be executed if the object is restrained. This is in fact a planning problem, and the solution will depend on the action instances added to the structure. The user would expand the initial structure by at least:

m **in** *move*
  $object(opponent) = true$

and

$u(X, Y)$ **in** *unrestrain*
  $agent(X) = true$
  $object(Y) = true$

After adding a planning module (Gelfond and Khal 2014) to the ASP translation of this extended system description, two possible solutions would be found: either that the wrestler unrestrains his opponent (i.e., execute $u(wrestler, opponent)$) or that the opponent unrestrains himself (i.e., execute $u(opponent, opponent)$). Note that the KM inference engine cannot perform planning.

6 Conclusions and future work

In this paper we have described a library of core commonsense knowledge about dynamic domains in modular action language $\mathcal{ALM}$. The library, called CoreALMLib, is obtained by translating a big part of an established upper ontology, CLib, into $\mathcal{ALM}$. We have provided a translation from the language of CLib into $\mathcal{ALM}$, thus narrowing the gap between different KR languages. We have discussed the methodology used to group resulting declarations and axioms into $\mathcal{ALM}$ modules. We have created an online tool that allows users to view the module dependency

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

hierarchy and search for relevant modules based on English verbs or adjectives. Finally, we have discussed how the library can be used in practice to reason about dynamic domains.

We believe that COREALMLIB is, at least in part, more elaboration tolerant than its CLIB counterpart, specifically with respect to the description of action effects and preconditions. On the other hand, restrictions on the values of attributes of a class are more elegantly expressed in CLIB. This indicates possible syntactic additions to ALM. COREALMLIB can be seamlessly coupled with reasoning algorithms in ASP to solve complex tasks such as diagnosis (Balduccini and Gelfond 2003) and planning, that cannot be answered in the inference engine of CLIB.

COREALMLIB can also further motivate the research on libraries in ALM, including the structuring of knowledge; further means for finding relevant modules; and providing to the user information about the contents of a module. It can also drive the investigation on how to allow users to select only those pieces of a module (declarations or axioms) that are relevant to a specific dynamic domain, which we believe to be an interesting research question.

Acknowledgements

This research was funded in part by SRI International and Vulcan Inc. The author is also grateful to Michael Gelfond for his feedback.

References

BAKER, C. F., FILLMORE, C. J. AND LOWE, J. B. 1998. The Berkeley FrameNet Project. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*, Association for Computational Linguistics and Morgan Kaufmann Publishers, San Mateo, CA, 86–90.

BALDUCCINI, M. 2013. ASP with non-Herbrand partial functions: a language and system for practical use. *Theory and Practice of Logic Programming 13,* 4–5, 547–561.

BALDUCCINI, M. AND GELFOND, M. 2003. Diagnostic Reasoning with A-Prolog. *Theory and Practice of Logic Programming 3,* 4–5 (Jul), 425–461.

BARKER, K., PORTER, B. AND CLARK, P. 2001. A library of generic concepts for composing knowledge bases. In *Proceedings of the First International Conference on Knowledge Capture*. K-CAP '01, ACM, New York, NY, USA, 14–21.

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.

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. Tech. rep., SRI International.

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 Fourth International Conference on Knowledge Capture*. K-CAP '07, ACM, New York, NY, USA, 159–166.

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

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press

posed to a knowledge-based system. In *Proceedings of Fourth International Conference on Knowledge Capture*. K-CAP '07, ACM, New York, NY, USA.

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.

Erdoğan, S. and Lifschitz, V. 2006. Actions as special cases. In *Principles of Knowledge Representation and Reasoning: Proceedings of the International Conference*, P. Doherty, J. Mylopoulos, and C. A. Welty, Eds. AAAI Press, Palo Alto, CA, 377–387.

Erdoğan, S. T. 2008. A Library of General-Purpose Action Descriptions. Ph.D. thesis, University of Texas at Austin, Austin, TX, USA.

Gelfond, M. and Khal, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, New York, NY, USA.

Gelfond, M. and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In *Proceedings of the International Conference on Logic Programming (ICLP'1988)*. MIT Press, Cambridge, MA, 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.

Gelfond, M. and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming 14,* 4–5, 587–601.

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. 2015. A CLib-inspired library of commonsense knowledge in modular action language ALM. In *Proceedings of the AAAI 2015 Spring Symposium on Formalizing Commonsense*. AAAI Press, Palo Alto, CA, 82–88.

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*. AAAI Press, Palo Alto, CA, 49–55.

Inclezan, D. and Gelfond, M. 2016. Modular Action Language ALM. *Theory and Practice of Logic Programming 16,* 2, 189–235.

Kipper-Schuler, K. 2005. VerbNet: A broad-coverage, comprehensive verb lexicon. Ph.D. thesis, University of Pennsylvania.

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

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D. and Miller, K. 1990. WordNet: An on-line lexical database. *International Journal of Lexicography 3*, 235–244.

https://doi.org/10.1017/S1471068416000363 Published online by Cambridge University Press