Online appendix for the paper

Modular Action Language $\mathcal{ALM}$

published in Theory and Practice of Logic Programming

Daniela Inclezan

Department of Computer Science and Software Engineering, Miami University
(*e-mail:* inclezd@MiamiOH.edu)

Michael Gelfond

Department of Computer Science, Texas Tech University
(*e-mail:* michael.gelfond@ttu.edu)

submitted 1 December 2013; revised 15 October 2014, 6 April 2015; accepted 12 May 2015

Appendix A Grammar of $\mathcal{ALM}$

$\langle boolean \rangle$:- true | false
$\langle non\_zero\_digit \rangle$:- 1 | ... | 9
$\langle digit \rangle$:- 0 | $\langle non\_zero\_digit \rangle$
$\langle lowercase\_letter \rangle$:- a | ... | z
$\langle uppercase\_letter \rangle$:- A | ... | Z
$\langle letter \rangle$:- $\langle lowercase\_letter \rangle$ | $\langle uppercase\_letter \rangle$
$\langle identifier \rangle$:- $\langle lowercase\_letter \rangle$ | $\langle identifier \rangle \langle letter \rangle$ | $\langle identifier \rangle \langle digit \rangle$
$\langle variable \rangle$:- $\langle uppercase\_letter \rangle$ | $\langle variable \rangle \langle letter \rangle$ | $\langle variable \rangle \langle digit \rangle$
$\langle positive\_integer \rangle$:- $\langle non\_zero\_digit \rangle$ | $\langle positive\_integer \rangle \langle digit \rangle$
$\langle integer \rangle$:- 0 | $\langle positive\_integer \rangle$ | $-\langle positive\_integer \rangle$
$\langle arithmetic\_op \rangle$:- + | − | ∗ | / | mod
$\langle comparison\_rel \rangle$:- > | >= | < | <=
$\langle arithmetic\_rel \rangle$:- $\langle eq \rangle$ | $\langle neq \rangle$ | $\langle comparison\_rel \rangle$
$\langle basic\_arithmetic\_term \rangle$:- $\langle variable \rangle$ | $\langle identifier \rangle$ | $\langle integer \rangle$
$\langle basic\_term \rangle$:- $\langle basic\_arithmetic\_term \rangle$ | $\langle boolean \rangle$
$\langle function\_term \rangle$:- $\langle identifier \rangle \langle function\_args \rangle$
$\langle function\_args \rangle$:- $(\langle term \rangle \langle remainder\_function\_args \rangle)$
$\langle remainder\_function\_args \rangle$:- ϵ | , $\langle term \rangle \langle remainder\_function\_args \rangle$
$\langle arithmetic\_term \rangle$:- $\langle basic\_arithmetic\_term \rangle \langle arithmetic\_op \rangle \langle basic\_arithmetic\_term \rangle$
$\langle term \rangle$:- $\langle basic\_term \rangle$ | $\langle arithmetic\_term \rangle$
$\langle positive\_function\_literal \rangle$:- $\langle function\_term \rangle$ | $\langle function\_term \rangle \langle eq \rangle \langle term \rangle$
$\langle function\_literal \rangle$:- $\langle positive\_function\_literal \rangle$ | $\neg \langle function\_term \rangle$ |
$\langle function\_term \rangle \langle neq \rangle \langle term \rangle$
$\langle literal \rangle$:- $\langle function\_literal \rangle$ | $\langle arithmetic\_term \rangle \langle arithmetic\_rel \rangle \langle arithmetic\_term \rangle$

⟨*var_id*⟩ :- ⟨*variable*⟩ | ⟨*identifier*⟩
⟨*body*⟩ :- ϵ | , ⟨*literal*⟩⟨*body*⟩
⟨*dynamic_causal_law*⟩ :- occurs(⟨*var_id*⟩) causes ⟨*positive_function_literal*⟩ if
instance(⟨*var_id*⟩, ⟨*var_id*⟩)⟨*body*⟩.
⟨*state_constraint*⟩ :- ⟨*sc_head*⟩ if ⟨*body*⟩.
⟨*sc_head*⟩ :- false | ⟨*positive_function_literal*⟩
⟨*definition*⟩ :- ⟨*function_term*⟩ if ⟨*body*⟩.
⟨*executability_condition*⟩ :- imposible occurs(⟨*var_id*⟩) if
instance(⟨*var_id*⟩, ⟨*var_id*⟩)⟨*extended_body*⟩.
⟨*extended_body*⟩ :- ϵ | , ⟨*literal*⟩⟨*body*⟩ | , occurs(⟨*var_id*⟩)⟨*extended_body*⟩ |
, ¬occurs(⟨*var_id*⟩)⟨*extended_body*⟩
⟨*system_description*⟩ :- system description ⟨*identifier*⟩ ⟨*theory*⟩⟨*structure*⟩
⟨*theory*⟩ :- theory ⟨*identifier*⟩⟨*set_of_modules*⟩ | import ⟨*identifier*⟩ from ⟨*identifier*⟩
⟨*set_of_modules*⟩ :- ⟨*module*⟩⟨*remainder_modules*⟩
⟨*remainder_modules*⟩ :- ϵ | ⟨*module*⟩⟨*remainder_modules*⟩
⟨*module*⟩ :- module ⟨*identifier*⟩⟨*module_body*⟩ |
import ⟨*identifier*⟩.⟨*identifier*⟩ from ⟨*identifier*⟩
⟨*module_body*⟩ :- ⟨*sort_declarations*⟩⟨*constant_declarations*⟩⟨*function_declarations*⟩⟨*axioms*⟩
⟨*sort_declarations*⟩ :- ϵ | sort declarations ⟨*one_sort_decl*⟩⟨*remainder_sort_declarations*⟩
⟨*remainder_sort_declarations*⟩ :- ϵ | ⟨*one_sort_decl*⟩⟨*remainder_sort_declarations*⟩
⟨*one_sort_decl*⟩ :- ⟨*identifier*⟩⟨*remainder_sorts*⟩ :: ⟨*sort_name*⟩⟨*remainder_sort_names*⟩⟨*attributes*⟩
⟨*remainder_sorts*⟩ :- ϵ | , ⟨*identifier*⟩⟨*remainder_sorts*⟩
⟨*remainder_sort_names*⟩ :- ϵ | , ⟨*sort_name*⟩⟨*remainder_sorts*⟩
⟨*sort_name*⟩ :- ⟨*identifier*⟩ | [⟨*integer*⟩..⟨*integer*⟩]
⟨*attributes*⟩ :- ϵ | attributes ⟨*one_attribute_decl*⟩⟨*remainder_attribute_declarations*⟩
⟨*one_attribute_decl*⟩ :- ⟨*identifier*⟩ : ⟨*arguments*⟩⟨*identifier*⟩
⟨*arguments*⟩ :- ϵ | ⟨*identifier*⟩⟨*remainder_args*⟩ →
⟨*remainder_args*⟩ :- ϵ | × ⟨*identifier*⟩⟨*remainder_args*⟩
⟨*remainder_attribute_declarations*⟩ :- ϵ |
⟨*one_attribute_decl*⟩⟨*remainder_attribute_declarations*⟩
⟨*constant_declarations*⟩ :- ϵ | object constants ⟨*one_constant_decl*⟩⟨*remainder_constant_declarations*⟩
⟨*one_constant_decl*⟩ :- ⟨*identifier*⟩⟨*constant_params*⟩ : ⟨*identifier*⟩
⟨*remainder_constant_declarations*⟩ :- ϵ | ⟨*one_constant_decl*⟩⟨*remainder_constant_declarations*⟩
⟨*constant_params*⟩ :- (⟨*identifier*⟩⟨*remainder_constant_params*⟩)
⟨*remainder_constant_params*⟩ :- ϵ | , ⟨*identifier*⟩⟨*remainder_constant_params*⟩
⟨*function_declarations*⟩ :- ϵ | function declarations ⟨*static_declarations*⟩⟨*fluent_declarations*⟩
⟨*static_declarations*⟩ :- ϵ | statics ⟨*basic_function_declarations*⟩⟨*defined_function_declarations*⟩
⟨*fluent_declarations*⟩ :- ϵ | fluents ⟨*basic_function_declarations*⟩⟨*defined_function_declarations*⟩
⟨*basic_function_declarations*⟩ :- ϵ | basic ⟨*one_function_decl*⟩⟨*remainder_function_declarations*⟩
⟨*defined_function_declarations*⟩ :- ϵ | defined ⟨*one_function_decl*⟩⟨*remainder_function_declarations*⟩
⟨*one_function_decl*⟩ :- ⟨*total_partial*⟩⟨*one_f_decl*⟩
⟨*total_partial*⟩ :- ϵ | total
⟨*one_f_decl*⟩ :- ⟨*identifier*⟩ : ⟨*identifier*⟩⟨*remainder_args*⟩ → ⟨*identifier*⟩
⟨*remainder_function_declarations*⟩ :- ϵ | ⟨*one_function_decl*⟩⟨*remainder_function_declarations*⟩

⟨*axioms*⟩ :- ϵ | axioms ⟨*one_axiom*⟩⟨*remainder_axioms*⟩
⟨*one_axiom*⟩ :- ⟨*dynamic_causal_law*⟩ | ⟨*state_constraint*⟩ | ⟨*definition*⟩ | ⟨*executability_condition*⟩
⟨*remainder_axioms*⟩ :- ϵ | ⟨*axiom*⟩⟨*remainder_axioms*⟩
⟨*structure*⟩ :- structure ⟨*identifier*⟩⟨*constant_defs*⟩⟨*instance_defs*⟩⟨*statics_defs*⟩
⟨*constant_defs*⟩ :- ϵ | constants ⟨*one_constant_def*⟩⟨*remainder_constant_defs*⟩
⟨*one_constant_def*⟩ :- ⟨*identifier*⟩ = ⟨*value*⟩
⟨*value*⟩ :- ⟨*identifier*⟩ | ⟨*boolean*⟩ | ⟨*integer*⟩
⟨*remainder_constant_defs*⟩ :- ϵ | ⟨*one_constant_def*⟩⟨*remainder_constant_defs*⟩
⟨*instance_defs*⟩ :- ϵ | instances ⟨*one_instance_def*⟩⟨*remainder_instance_defs*⟩
⟨*one_instance_def*⟩ :- ⟨*object_name*⟩⟨*remainder_object_names*⟩ in
⟨*identifier*⟩⟨*cond*⟩⟨*attribute_defs*⟩
⟨*object_name*⟩ :- ⟨*identifier*⟩⟨*object_args*⟩
⟨*object_args*⟩ :- ϵ | (⟨*basic_term*⟩⟨*remainder_object_args*⟩)
⟨*remainder_object_args*⟩ :- ϵ | , ⟨*basic_term*⟩⟨*remainder_object_args*⟩
⟨*remainder_object_names*⟩ :- ϵ | , ⟨*object_name*⟩⟨*remainder_object_names*⟩
⟨*cond*⟩ :- ϵ | where ⟨*literal*⟩⟨*remainder_cond*⟩
⟨*remainder_cond*⟩ :- ϵ | , ⟨*literal*⟩⟨*remainder_cond*⟩
⟨*attribute_defs*⟩ :- ϵ | ⟨*one_attribute_def*⟩⟨*remainder_attribute_defs*⟩
⟨*one_attribute_def*⟩ :- ⟨*identifier*⟩⟨*object_args*⟩ = ⟨*basic_term*⟩
⟨*statics_defs*⟩ :- ϵ | values of statics ⟨*one_static_def*⟩⟨*remainder_statics_defs*⟩
⟨*one_static_def*⟩ :- ⟨*function_literal*⟩ if ⟨*body*⟩.
⟨*remainder_statics_defs*⟩ :- ϵ | ⟨*one_static_def*⟩⟨*remainder_statics_defs*⟩

Appendix B $\mathcal{ALM}$ and the Digital Aristotle

The reader may have noticed that the $\mathcal{ALM}$ 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 $\mathcal{ALM}$. In this section, we show how the reuse of knowledge in $\mathcal{ALM}$ 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 $\mathcal{ALM}$'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 $\mathcal{ALM}$, 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.[1] 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 $\mathcal{ALM}$ 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 $\mathcal{ALM}$ 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

[1] http://www.allenai.org/TemplateGeneric.aspx?contentId=9

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 $\mathcal{ALM}$ 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 $\mathcal{ALM}$ 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 N^{th} 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
        $length : positive\_natural\_numbers$
        $component : [0..length] \rightarrow universe$
    $action\_sequences :: sequences$
  axioms
    $false$ **if** $component(S, N) = E,$
    $instance(S, action\_sequences),$
    $\neg instance(E, actions),$
    $\neg instance(E, action\_sequences).$

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 pro-

cess, 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_{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, $num : types\_of\_parts \times types\_of\_parts \rightarrow natural\_numbers$, where $num(P_1, P_2) = N$ holds if the number of elements of type P_1 in one element of type P_2 is N. For instance, $num(nucleus, cell) = 2$ 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: *duplicate* and *split*. *Duplicate*, which acts upon an *object* that is an element from sort *types_of_parts*, doubles the number of every part of this kind present in the environment. *Split* also acts upon an *object* ranging over *types_of_parts*. An action a of this type with $object(a) = c_1$, where c_2 is a child of c_1 in H_{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 *split* with $object(a) = cell$ 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, *prevent_duplication*, with an attribute *object* with the range *types_of_parts*. The occurrence of an instance action a of *prevent_duplication* with $object(a) = c$ 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* × *types_of_parts* → *booleans*

fluents
basic
total $num : types\_of\_parts \times types\_of\_parts \rightarrow natural\_numbers$
$prevented\_dupl : types\_of\_parts \rightarrow booleans$

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** $is\_part\_of(P_1) = P2$.

$part\_of(P_1, P_2)$ **if** $is\_part\_of(P_1) = P_3$, $part\_of(P_3, P_2)$.

$num(P, P) = 0$.

$num(P_3, P_1) = N$ **if** $is\_part\_of(P_3) = P_2$, $part\_of(P_2, P_1)$, $num(P_2, P_1) = N_1$, $num(P_3, P_2) = N_2$, $N_1 * N_2 = N$.

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 1^{st} component of *cell_cycle* is *interphase*" and "the 2^{nd} 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. $\mathcal{ALM}$ 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 $\mathcal{ALM}$'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 imported

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

$$
\begin{array}{l}
observed(num(cell, sample), 1, 0) \\
observed(num(nucleus, cell), 1, 0) \\
intend(cell\_cycle, 0) \\
\neg happened(cytokinesis, I)
\end{array}
$$

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:

$$
\begin{array}{lcl}
answer(X, \text{“cells per sample”}) & \leftarrow & last\_step(I), \\
 & & num(cell, sample, I) = X. \\
answer(X, \text{“nuclei per cell”}) & \leftarrow & last\_step(I), \\
 & & num(nucleus, cell, I) = X.
\end{array}
$$

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 $cell\_cycle(2)$; 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:

$intend(cytokinesis, 2)$ $\quad$ $\neg occurs(cytokinesis, 2)$
$intend(cytokinesis, 3)$ $\quad$ $\neg occurs(cytokinesis, 3)$
$intend(cytokinesis, 4)$ $\quad$ $\neg occurs(cytokinesis, 4)$
...

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

$answer(1,$ "cells per sample"$)$ $\quad$ $holds(val(num(cell, sample), 1), 2)$
$answer(2,$ "nuclei per cell"$)$ $\quad$ $holds(val(num(nucleus, sample), 2), 2)$
$last\_step(2)$ $\quad$ $holds(val(num(nucleus, cell), 2), 2)$

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 $\mathcal{ALM}$ and MAD

In this section we give an informal discussion of the relationship between $\mathcal{ALM}$ and the modular action language 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 $\mathcal{ALM}$ and MAD. The former is a modular expansion of action language $\mathcal{AL}$. The latter expands action language $\mathcal{C}$ (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 $\mathcal{AL}$ incorporates the Inertia Axiom, which says that "*Things normally stay the same.*" Language $\mathcal{C}$ 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. 2004). In $\mathcal{C}$ the inertia axiom for a literal l is expressed by a statement

caused l **if** l **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 $\mathcal{AL}$ allows two types of fluents – inertial and defined –, $\mathcal{C}$ 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 $\mathcal{AL}$ or $\mathcal{ALM}$. Of course, the question is not settled and our opinion can change with additional experience. On another hand, $\mathcal{AL}$ allows recursive state constraints and definitions, which are severely limited in $\mathcal{C}$. There is a close relationship between ASP and $\mathcal{C}$ but, in our judgment, the distance between ASP and $\mathcal{AL}$ is smaller than that between ASP and $\mathcal{C}$. There is also a substantial difference between modules of $\mathcal{ALM}$ and MAD.

To better understand the relationship let us consider the $\mathcal{ALM}$ theory *motion* and the system description *travel* from Section 3.2 and represent them in MAD.[2]

Example 1 (*A MAD Version of the System Description* travel)
The $\mathcal{ALM}$ system description *travel* is formed by the theory *motion* and the structure *Bob_and_John*. The theory consists of two modules, *moving* and *carrying_things*, organized into a module hierarchy in which the latter module depends on the former. Let us start with the MAD representation of $\mathcal{ALM}$'s module *moving*.

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

[2] Although the "Monkey and Banana" problem presented in Section 4.1 has been encoded in 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 $\mathcal{ALM}$ versus MAD from the knowledge representation point of view.

will be 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 $M_1 = MAD(moving)$ 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 $\mathcal{ALM}$):

sorts
$Universe; Points; Things; Agents;$
inclusions
$Points << Universe;$
$Things << Universe;$
$Agents << Things;$

The **sorts** part declares the sort *universe* (which is pre-defined and does not require declaration in $\mathcal{ALM}$) 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 (Erdoğan 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 $\mathcal{ALM}$.

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 $\mathcal{ALM}$ 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
$t, t_1, t_2 : Things$;
$actor : Agents$;
$origin, dest : Points$;
axioms
inertial $Loc\_in(t)$;
exogenous $Move(actor, dest, origin)$;

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

$Move(actor, dest, origin)$ **causes** $Loc\_in(actor) = dest$;

Similarly for the executability conditions:

nonexecutable $Move(actor, dest, origin)$ **if** $Loc\_in(actor) \neq origin$;
nonexecutable $Move(actor, dest, origin)$ **if** $Loc\_in(actor) = dest$;
nonexecutable $Move(actor, dest, origin)$ **if** $Loc\_in(actor) = origin$, $\neg Connected(origin, dest)$;

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 M_2 that corresponds to *carrying_things* contains declarations of the new action $Carry$ and the corresponding variables.

module M_2;
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 $\mathcal{ALM}$ 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:

import M_1;
$Move(actor, dest, origin)$ **is** $Carry(actor, carried\_object, dest, origin)$;

Intuitively, this import statement says that the action $Carry(actor, carried\_object, dest, origin)$ has all properties that are postulated for the action $Move(actor, dest, origin)$ 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 *carrying_things*:

axioms
exogenous $Carry(actor, carried\_object, dest, origin)$;

% State constraints:
$Is\_held(c)$ **if** $Holding(t, c)$;

% Executability conditions:
nonexecutable $Carry(actor, carried\_object, dest, origin)$ **if**
$\neg Holding(actor, carried\_object)$;
nonexecutable $Move(actor, dest, origin)$ **if** $Is\_held(actor)$;

Note, however, that the $\mathcal{ALM}$ module *carrying_things* also contained the *recursive* state constraints below, saying that agents and the objects they are holding have the same location:

$loc\_in(C) = P$ **if** $holding(T, C), loc\_in(T) = P$.
$loc\_in(T) = P$ **if** $holding(T, C), loc\_in(C) = P$.

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

$Move(actor, dest, origin)$ **causes** $Loc\_in(c) = dest$ **if** $Holding(actor, c)$;

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 *carry* in a new module that we call M_2, in which we import module M_1 while renaming $Move(actor, dest, origin)$ as $Carry(actor, carried\_object, dest, origin)$. In $\mathcal{ALM}$ the declarations of *move* and its specialization *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 $\mathcal{ALM}$ representations.

Finally, we consider the structure of our $\mathcal{ALM}$ system description. It contains two types of actions $go(Actor, Dest)$ and $go(Actor, Dest, Origin)$. 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 M_1;
$Move(actor, dest, origin)$ **is** $Go(actor, dest, origin)$;

import M_1;
$Move(actor, dest, origin)$ **is** $Go(actor, dest)$;

import M_2;
$Carry(actor, Suitcase, dest, origin)$ **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 $\mathcal{ALM}$ and MAD. Here is a short summary:

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

  $$p \textbf{ if } p.$$

  is not equivalent to the same axiom in MAD. The $\mathcal{ALM}$ 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 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 MAD counterpart of the $\mathcal{ALM}$ theory *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 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 *actions* is part of the sort hierarchy, whereas in MAD actions are not considered sorts. Instead, MAD has special constructs **import** and **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 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 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 *actions* sort, for instance:
  $$intended : agent\_actions \rightarrow booleans$$
  where *agent_actions* is a special case of *actions*. There is no equivalent concept in MAD, where fluents must be defined on, and range over, either primitive sorts or the built-in sort *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 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 MAD*
  In 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 $\mathcal{ALM}$. 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 $\mathcal{ALM}$ 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 $\mathcal{ALM}$*
  Some axioms, allowed in MAD, are not directly expressible in $\mathcal{ALM}$. For instance, MAD axioms of the type:

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

  or

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

  belong to this group. The first axiom allows to specify non-deterministic effects of actions, while the second assignes 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 $\mathcal{ALM}$. Non-determinism, however, is an important feature that one should be able to express in an action formalism. It may be added to $\mathcal{ALM}$ (and to $\mathcal{AL}$) in a very natural manner, but it is not allowed in $\mathcal{AL}$ and the mathematical properties of "non-deterministic" $\mathcal{AL}$ were not yet investigated. Because of this we decided to add this feature in the next version of $\mathcal{ALM}$.

We hope that this section gives the reader some useful insight in differences between $\mathcal{ALM}$ and MAD. We plan to extend the comparison between $\mathcal{ALM}$ 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 $\mathcal{ALM}$ to action descriptions of MAD can consult (Inclezan 2012).

References

Baral, C. and Gelfond, M. 2005. Reasoning about Intended Actions. In *AAAI-05: Proceedings of the 20th National Conference on Artificial Intelligence*. AAAI Press, 689–694.

Campbell, N. A. and Reece, J. B. 2001. *Biology*, 6th ed. Benjamin Cummings.

Erdoğan, S. and Lifschitz, V. 2006. Actions as special cases. In *Principles of Knowledge Representation and Reasoning: Proceedings of the International Conference*. 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 Lifschitz, V. 2012. The Common Core of Action Languages B and C. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'2012)*.

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

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

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. 2012. Modular Action Language ALM for Dynamic Domain Representation. Ph.D. thesis, Texas Tech University, Lubbock, TX, USA.

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.