Undergraduate Honors Thesis:

Testing a library of general-purpose action descriptions

by

Ashu Manohar

Advisor: Vladimir Lifschitz

Summer 2006

Abstract

Reasoning about actions and describing changes caused by the execution of these actions is an idea central to common sense reasoning. Action description languages have been developed to specify the effects and preconditions of actions using a logical framework. However, many of these formalisms suffer from a lack of generality and modularity [McCarthy, 1987].

A new *M*odular *A*ction *D*escription language (MAD) [Lifschitz & Ren, 2006] and a library of general purpose action descriptions [Erdogan, unpublished] written in MAD are being designed to solve these problems. This thesis is the first step towards testing the semantics of MAD and the ability of this library to succinctly and expediently describe new action domains.

1. Introduction

One of the major long term goals of Artificial Intelligence (AI) research is to endow computers with commonsense. One of the earliest suggestions on dealing with commonsense problems was to use formal logic. The roots of this branch of AI, known as logical AI, are found in the 1959 paper by McCarthy – Programs with Common Sense [McCarthy, 1959]. In this paper, he described that *"a program has common sense if it automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told from what it already knows."*

Logical AI provides an axiomatic basis for reasoning about the world inhabited by agents who have beliefs and goals, who perform actions in order to reach these goals and by doing so change the state of the world. The agent infers a course of actions that will lead to its goals by performing logical analysis i.e. commonsense reasoning.

Reasoning about actions and describing changes caused by the execution of these actions is an idea central to commonsense reasoning. Action description languages – formal languages used to describe the effects of actions – have been developed to represent and solve problems in logical AI. Many of these formalisms suffer from a lack of generality and modularity [McCarthy, 1987]. A new *M*odular *A*ction *D*escriptive language (MAD*)* [Lifschitz & Ren, 2006] aims to solve one half of this problem by allowing re-use of existing action descriptions. The other half that is necessary is a database of general-purpose action descriptions which can be used to succinctly and expediently describe new action domains.

In this thesis, we describe our efforts to test the semantics of MAD and the applicability of such a library to describe new action domains [Erdogan, et al., 2006]. The *library* is an application of the idea of a database of reusable, general-purpose knowledge components [Clark and Porter, 1997; Porter, et al. 2001] to the design of action languages. It is being written in MAD*,* and is a topic of Selim Erdogan's dissertation proposal [Erdogan, unpublished]. The idea of such a library was first explored in [Erdogan & Lifschitz, 2006].

Different actions in different domains are often related. Many actions can be described in terms of other actions [Erdogan & Lifschitz, 2006] – for example, *pushing*, *carrying*, and *walking* can all be described as special cases of a more general action *move*. Action domains tend to use *specialized* versions of such general actions, and in the process of formalization, reinvent knowledge. This *library* aims to factor out common aspects of these *specialized* actions to form a repository of reusable knowledge components, which will contain descriptions of general actions such as *move*. A formalization of an action domain could then *"import"* general knowledge from this library, with very few domain-specific axioms needed.

To draw an analogy, this library is very similar to libraries (API) available in widely-used procedural languages such as C++ [Stroustrup, 2000] and Java [Arnold et al., 2000]. Differences exist in that this library represents knowledge as declarative axioms as compared to the imperative nature of programming in procedural languages.

The *library* and MAD are still in the developmental stages, and much work is needed before either is fully mature. The future implementation of MAD *(*MAD *Causal Calculator)* is still in the works. Consequently, we have no means to verify the correctness, completeness, or quality of the library in achieving its goals at present. To solve this problem, we have developed a method of translating the *library* and a MAD formalization (of an action domain) into the language of the *Causal Calculator*[1] *(CCalc)*, and testing it using CCALC.

In section 2 we give a brief background of the development of action description languages, MAD, and an example; in section 3 we describe the semantics of MAD; in section 4 we explain our algorithm which translates MAD action descriptions to input for CCalc; in section 5 we discuss our

[1] http://www.cs.utexas.edu/users/tag/cc/

results – the errors we found on testing the library and our solutions; (Appendix A) contains segments of code used by our algorithm.

2. Background

Common sense capability can be broadly divided into common sense knowledge and common sense reasoning. Common sense knowledge concerns situations that change in time as a result of events and their effects. The most important events are actions. Commonsense reasoning involves drawing conclusions based on this knowledge. Facts about the world can be represented as logical axioms and deduction methods can be used to reason about the changes in the states of the world [McCarthy, 1959].

Planning problems provide one of the most fruitful showcases for combining logical analysis with AI applications. On the one hand there are many practically important applications of automated planning, and on the other logical formalizations of planning are genuinely helpful in understanding the problems and in designing algorithms. In such a problem, an agent in an initial-world state is equipped with a set of *actions*, which can be thought of as partial functions transforming world states into world states. These actions are feasible only in world states that meet certain constraints (called "pre-conditions" of the action). The goal of the agent is to find a sequence of feasible actions that take it from the initial state to the desired state.

Important problems, such as the *frame problem* [McCarthy, 1979] and *ramification problem* [Finger, 1986] have arisen in the research of logical AI. They have been successfully solved using nonmonotonic formalisms [Shanahan, 1997; Geffner, 1990; Lin, 1995; McCain and Turner, 1997]. In particular, very expressive action descriptive languages have been introduced, incorporating the solutions of these problems in their design. They define "transition systems" – directed graphs with the vertices representing all the possible states of an action domain and the edges representing actions an agent can take.

A rich and eclectic set of conceptual tools has transformed the study of logical AI. This process and its outcome are well documented in [Russell & Norvig 2003]. However, work is still needed to make these systems "generally" applicable. In reviewing his Turing Award lecture of 1971 [McCarthy, 1987], McCarthy notes –

> It was obvious in 1971 and even in 1958 that AI programs suffered from a lack of generality. It is still obvious, and now there are many more details. *The first gross*

symptom is that a small addition to the idea of a program often involves a complete rewrite beginning with the data structures. Some progress has been made in modularizing data structures, but small modifications of the search strategies are even less likely to be accomplished without rewriting.

Another symptom is that *no-one knows how to make a general database of common sense knowledge that could be used by any program that needed the knowledge*. Along with other information, such a database would contain what a robot would need to know about the effects of moving objects around, what a person can be expected to know about his family, and the facts about buying and selling. This doesn't depend on whether the knowledge is to be expressed in a logical language or in some other formalism. When we take the logic approach to AI, lack of generality shows up in that the axioms we devise to express common sense knowledge are too restricted in their applicability for a general common sense database. In my opinion, getting a language for expressing general common sense knowledge for inclusion in a general database is the key problem of generality in AI.

MAD [Lifschitz and Ren, 2006] aims to partially solve this issue. The "high-level" notation of MAD along with the availability of transition system semantics, make it an attractive formalism for describing actions. The semantics of MAD is based on *C*+. The main distinctive feature of MAD is its use of modules to describe an action domain. Each module describes a set of interrelated fluents[2] and actions. A module may contain references to other modules via *import* statements, which allows a new action domain to refer to other previously defined action descriptions. These *import* statements also allow the user to *specialize* or *rename* the general actions and fluents.

MAD gives us the ability to build a hierarchy of action descriptions and inherit properties from parent modules. So, MAD is an appropriate language for developing the library. To quote from [Erdogan, unpublished] –

The library will consist of MAD modules, each describing a group of general commonsense facts related to actions. For example, one module might describe the

[2] A fluent is a condition that changes over time. Fluents are generally represented by predicates having an argument that depends on time. However, a fluent can also be represented as a function. For example, *Location*(*Thing*) is a function representation of a fluent that maps *things* to *places*.

effects of the “move” action, including the axiom “moving an object causes it to be at a new location.” Another module might express more general information about locality, such as “an agent must be at the same location as an object to be able to perform an action on it.”

The Causal Calculator (CCalc) is an implementation of a subset of *C+*, which can be used to solve problems in commonsense reasoning, such as prediction and planning. It has been applied to several challenging problems in commonsense reasoning [Lifschitz et al., 2000, Lifschitz, 2000, Campbell and Lifschitz, 2003, Akman et al., 2004], including domains of non-trivial size. We intend to use *CCalc* to test the library and a MAD formalization after translating the latter to the language of *CCalc*.

2.1. Example

The following figure (Fig. 1) illustrates the use of the MAD language and the library to model a simplified version of the *Monkey and Bananas* (*M & B*) domain[3]. In this version (Monkey and Box), there are no bananas and the monkey’s goal is to simply climb the box. This is a typical example of a planning problem – a search for a series of executable actions that successively transform the initial state of the world into the desired state of the world.

This action description consists of two modules – a library module *MOVE* and a domain specific module *MONKEY*. Module *MOVE* is an axiomatization of “move-like” actions, which cause the location of the thing being moved to change. This idea is embedded in the axiom

Move(x, p) **causes** *Location(x) = p;*

which forms the crux of this module. Module *MONKEY imports* module *MOVE* twice, to describe two separate actions – walking and climbing i.e. horizontal movement and vertical movement. Action *walk* is defined using the first import statement. It is understood as a special case of action *Move*, wherein the *monkey* is the only *thing* that can move. Action *ClimbOn* is defined using the second import statement. It is also understood as a special case of action *move*, but in this case the *monkey moves* to a different level – from the *floor* onto the top of the *box*.

[3] A monkey tries to grab a bunch of bananas that is hanging from the ceiling and out of its reach. It can grab the bananas by pushing a box to the empty place under the bananas and climbing on top of the box.

Refer to [Erdogan, unpublished] for the complete action description of *M & B*, and other toy-domains.

```
module MOVE;
  sorts
    Thing; Place;
  constants
    Location(Thing): fluent(Place);
    Move(Thing, Place): action;
  variables
    x: Thing; p: Place;
  axioms
    inertial Location(x);
    exogenous Move(x, p);
    Move(x, p) causes Location(x) = p;
    nonexecutable Move(x, p) if Location(x) = p;
endmodule;

module MONKEY;
  sorts
    Thing; Place; Level ;
  objects
    Monkey, Box : Thing;
    P1, P2: Place;
    BoxTop, Floor : Level ;
  constants
    OnBox : fluent;
    Walk(Place), ClimbOn, ClimbOff : action;
variables
  x: Thing; p: Place; l: Level ;

import MOVE;
  Move(x, p) is Walk(p) ∧ x = Monkey;

import MOVE;
  Place is Level ;
  Location(x) = l is
      ((x = Monkey ∧ OnBox ) ∧ l = BoxTop) ∨
      (¬(x = Monkey ∧ OnBox ) ∧ l = Floor );
  Move(x, l) is
      (x = Monkey ∧ l = BoxTop ∧ ClimbOn) ∨
      (x = Monkey ∧ l = Floor ∧ ClimbOff );

axioms
  Location(Monkey) = p
  if OnBox ∧ Location(Box) = p;
      nonexecutable ClimbOn
  if Location(Monkey) ≠ Location(Box);
      nonexecutable ClimbOff ∧ Walk(p);

  endmodule
```

Figure 1: Example – Monkey and Box action description in MAD

In the initial state, the monkey is at location P1, and the box and bananas at P2. The correct two-step solution for the monkey to reach its goal entails the following actions:

1.*walk*(*P2*)

2.*ClimbOn*

In the example in section 3.1, we further explain the mechanics of the first import statement.

2.2. Library Prototype

Our current prototype of the library [Erdogan, unpublished] is a collection of fifteen modules that encapsulate the following concepts –

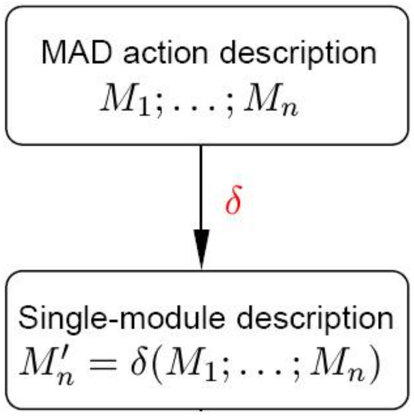
- Actors and Themes – Modules about actions in general. For example, an action maybe executed by an "actor" (performing agent like the *monkey* above). An action may also have a "theme" associated i.e. the thing the action affects (example: *theme* of action *Walk* is the *Monkey* itself). There may be zero or more themes for an action.
- Places and Movement – Describe movement, such as *climbing on* the box or *walking*. In order to formalize such actions, we need to express where things are located and how an action changes the location of that thing.
- Supporting Objects – Modules that express how things are supported, and how actions affect the *supporters*. For example, the monkey is either supported by the *floor* and the *box* through time, and *climbing* changes the *thing* that supports the *monkey* (from *floor* to *box*).
- General Properties of Actions – Represent preconditions of actions, like non-locality. An *agent* can't perform an *action* on a *thing* unless it is next to it.

The initial idea for building this library was to study many different domains in order to assemble a large number of interesting and useful library modules. However, the Monkey and Bananas (M & B) domain was found to be incredibly rich in itself, when viewed with an eye towards distinguishing generalizable features. Our current version of the library is based off such features in M & B.

This version of the library has been used to formalize other toy-domains [Erdogan, unpublished]. In the future, we believe that our library will grow us we study more domains and include general features from those domains.

3. Semantics of *M*odular *A*ction *D*escription (MAD) language[4]

3.1. Overview

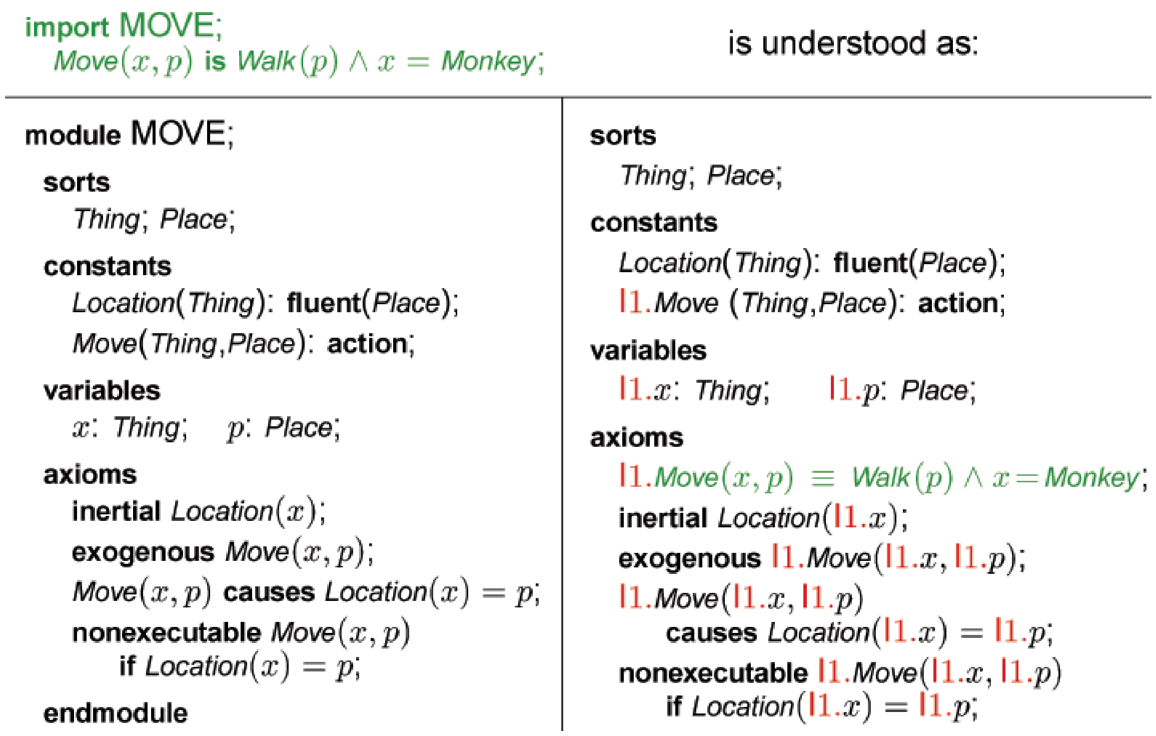


- Function δ turns an arbitrary MAD action description into a single module by replacing every import statement with a modified copy of the corresponding module.
- A single-module action description in MAD is essentially an action description in *C*+.

Example

Fig. 2 illustrates the effect of applying function δ on the first *import* statement in the Monkey & Box example we encountered in Fig. 1. This *import* statement demonstrates the *renaming* of action *Move* to *Walk* and is understood in this context as the *monkey* walking to a *place*. On the left is the original version of module *MOVE* and on the right is the modified copy.

[4] Please refer to [Lifschitz & Ren, 2006] for a thorough description


import MOVE;
Move(x, p) **is** *Walk*$(p) \wedge x =$ *Monkey*;

is understood as:

module MOVE;
sorts
Thing; *Place*;
constants
Location(*Thing*): **fluent**(*Place*);
Move(*Thing*,*Place*): **action**;
variables
x: *Thing*; p: *Place*;
axioms
inertial *Location*(x);
exogenous *Move*(x, p);
Move(x, p) **causes** *Location*$(x) = p$;
nonexecutable *Move*(x, p)
if *Location*$(x) = p$;
endmodule

sorts
Thing; *Place*;
constants
Location(*Thing*): **fluent**(*Place*);
I1.*Move* (*Thing*,*Place*): **action**;
variables
I1.x: *Thing*; I1.p: *Place*;
axioms
I1.*Move*$(x, p) \equiv$ *Walk*$(p) \wedge x =$ *Monkey*;
inertial *Location*(I1.x);
exogenous I1.*Move*(I1.x, I1.p);
I1.*Move*(I1.x, I1.p)
causes *Location*(I1.x) = I1.p;
nonexecutable I1.*Move*(I1.x, I1.p)
if *Location*(I1.x) = I1.p;

Fig 2. Applying function δ on the first import statement of the Monkey & Box domain

3.2. Function δ – Generating a single-module description

Function δ translates an arbitrary MAD action description into a single module by replacing every import statement with a modified copy of the corresponding module. This single-module action description in MAD is essentially an action description in *C*+.

MAD offers two types of *specialization* or *renaming* as part of the import statements – sort *renaming* and constant *renaming*. There are rules with each flavor of *renaming*. Note, first, that any import statement has the following form

import *NAME* ;
s_1 **is** s'_1 ;
…
s_k **is** s'_k ;
c_1 (…) **is** F_1 ;
….
c_l (…) **is** F_l;

where *NAME* is a module name, $s_1, \ldots, s_k, s'_1, \ldots, s'_k$ are sort names, $c_1, \ldots, c_l$ are constant names, and $F_1, \ldots, F_l$ are predicate formulas. (The dots after each c_j represent optional arguments). The lines following the *import* statement are known as *renaming* clauses.

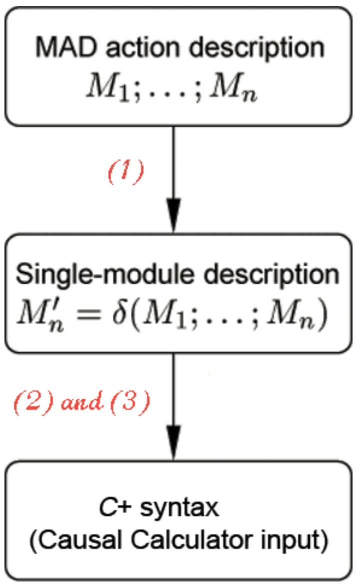
The rules of translation are –

- Variable *renaming* – always performed irrespective of any *renaming* clauses
    - Prepend *In* to each occurrence of every variable name (where *n* is the import number)
    - Example – x: *Thing* → I1.x: *Thing*
- Sort *renaming* (s_i **is** s_i') –
    - Replace every occurrence of each of sort name s_i with s'_i $(i = 1 \ldots k)$
- Constant *renaming* (c_j (...) **is** F_j)
    - Prepend *In* to every occurrence of this constant
    - Insert equivalences *In*.c_j (...) $\equiv F_j$ at the beginning of the axioms section. $(j = 1 \ldots l)$
    - Example –
        - Original: *Move(x, p)* **causes** *Location(x)* = *p* ;
        - Renamed: I1.*Move(x, p)* $\equiv$ *Walk(p)* $\wedge$ *x* = *Monkey* ;

          I1.*Move(*I1.*x,* I1. *p)* **causes** *Location(*I1.*x)* = I1. *p* ;

4. Translating the *library* and a MAD formalization into the language of the *Causal Calculator*

4.1. Overview of the Translation Process


MAD action description
$M_1; \ldots ; M_n$
(1)
Single-module description
$M'_n = \delta(M_1; \ldots ; M_n)$
(2) and (3)
C+ syntax
(Causal Calculator input)

Translating a MAD formalization to CCalc:

1. A Perl script implements the function δ, which implements the translation rules listed above. The result is a non-definite action description.
2. Translate non-definite axioms into definite[5].
3. Make textual substitutions so that the action description can be treated as an input to CCalc.

Currently, only step (1) of this process is automated through a Perl script. We're considering the automation of step 3.

Steps (1) and (2) will be performed by the future implementation of MAD *CCalc as well.*

4.2. Why Perl?

At this point, it is appropriate to discuss some of the reasons behind choosing Perl as the language to implement function δ. Perl [Wall, et. al., 2000] is a procedural programming language broadly based on C [Kernighan & Ritchie, 1988] with variables, control blocks, and subroutines. More importantly, for our purposes, it combines key features from *AWK* (associative arrays or "hash"s) [Aho, et. al., 1988] and *sed* (regular expressions) [Robbins & Dougherty, 1997].

The crux of implementing function δ lies in parsing *import* statements and the associated renaming clauses. This involves making non-trivial textual substitutions, which are handled

[5] The translation process and definite/non-definite laws are discussed in [Erdogan & Lifschitz, 2006].

well by Perl's regular expression engine. For example, the following single line Perl statement can trim the white-space from the beginning of a line of text.

```
$line =~ s/^\s+//;
```

It would take a few lines of code to perform the same in *C*.

Another Perl feature utilized by our algorithm is associative arrays. Perl includes this convenient data structure, which allows indexing of an array by strings rather than numbers. For example, the following command

```
$hash{"sorts"}
```

allows us to access an element indexed by the string "sorts".

4.3. Step 1: Perl script – Implementing the function δ

The script accepts one file as its input, which contains the MAD action descriptions in hierarchical order i.e. if a module M_1 imports another module M_2, then M_1 should be defined before M_2 in the file.

The script is comprised of two functions – the main body and a subroutine

%hash parse_module(@array);

The subroutine implements the crux of the algorithm – the rules related to *import* statements and *renaming* clauses. It accepts an unprocessed MAD module, and generates a single-module free of import statements. This single-module is then used as an input for step 2, which "definitizes" the action description.

4.3.1. Algorithm[6]

4.3.1.1. General outline –

- Without loss of generality, we can assume that each module can be broken up into the following sections. (We will refer to these as keywords.)

  sorts, objects, constants, variables, axioms, import

  We translate a whole module by parsing one section at a time. Each line in a module belongs to one of these sections.

[6] Segments of the Perl code for the algorithm can be found in Appendix A.

- A section can be further categorized into two blocks – *import* sections or *non-import* sections. The first five sections noted above are *non-import* sections, while the last is an *import* section. To apply the rules of translation, we iterate through the sections and process each line according to the block it belongs in i.e. all *import* sections are handled in one way and all *non-import* sections in another.
- Finally, we add the contents of the imported module to the module being processed and remove duplicate entries in the *sorts, constants*, etc. sections.
- **Data structures** – We use a *Hash of Arrays* to store the various sections of each module. By using a hash, we are assured amortized O(1) for add and access, the two functions we perform. Also, we make optimal use of space.
  Though this benefit in time and space complexity is small for our current needs, it might be useful when the grammar of MAD is expanded to include more sections.

Note: *$import_mod_hash{"sorts"}* denotes a pointer to an array that contains the "sorts" section of the module being referred to.

4.3.1.2. *Main* section

Separate the library modules, and send each to the subroutine *parse_module,* which in turn generates a single-module for each, free of *import* statements.

4.3.1.3. Subroutine *parse_module*

Parsing a section in the non-import sections block –

These sections are relatively easier to parse compared to the *import* sections.

1. Such a section begins with a line containing one of the keywords except *import*.
2. All the lines in this section are read into an array, and then a pointer to this array is stored in the *hash of arrays*.

Parsing an import section -

1. Prepend *In* to each occurrence of every variable name, where *n* is the import number.
2. Check for *renaming* clauses and handle each case separately.
    a) Sort *renaming* according to the rules stated in section 3.2 (Semantics of MAD).

   b) Constant *renaming* according to the rules stated in section 3.2 (Semantics of MAD).

3. Add the processed sections from the imported module to the original module and remove duplicates in the various sections. Duplicates can arise, for example, when a sort is implicitly declared in a module being *imported*, and also explicitly declared in the module itself.

4.4. Step 2: Converting to a definite theory

At this point, we have a single-module "non-definite" action description. Since, the Causal Calculator is an implementation of the "definite" fragment of *C+*, it is not possible to process this action description directly. However, [Erdogan & Lifschitz, 2006] propose several methods for "definitizing" an action description[7]. This step is currently performed by hand. Consider the following non-definite axiom from the example in section 3.1:

I1.$Move(x, p) \equiv Walk(p) \wedge x=Monkey;$

Note that *Move(x, p)* is an *exogenous* action, which satisfies a condition under which we can transform this law into a definite law. The following is an equivalent definite theory -

always I1.$Move(x, p) \equiv Walk(p) \wedge x=Monkey;$

exogenous *Walk(p);*

We prepended the keyword *always* and added an axiom defining *Walk* to be an *exogenous* action since action *Move* was.

4.5. Step 3: Textual substitutions in order to translate to CCalc input

After "definitizing", all we have left to do is make some textual substitutions to render the action description suitable for input to CCalc. This involves making trivial changes such as changing semi-colons to periods, changing the case of keywords, prepending static laws with the keyword *always*, etc. At present, we make each type of change individually using Perl on the command line. However, we plan to bunch these together in a Perl script.

After this step, we should have a single-module action description that can be run in CCalc.

[7] Note that it is not generally possible to "definitize" an action description. It is, however, in this case.

5. Results

In translating the library modules to *CCalc* semantics and testing the translated single module in *CCalc*, we found several bugs in the library and our MAD formalization of *Monkey & Bananas*. Upon correcting these errors, we were successfully able to run queries, which gave correct answers. This is an important milestone in the development of the library as were able to verify its correctness to model an action domain. We will now briefly discuss the errors found.

5.1. Error in function δ – semantics of MAD

The first and most important error was in the rules of generating a single-module outlined in section 3.2 – these rules are incomplete. As a result, there were multiple (different) declarations of a constant (I1.Move) in our single module. To solve this problem, we add another rule to the way constants are *renamed* while *importing* a module. This new rules is

- *Rename* all constants that have been previously *renamed*. In terms of implementing function δ, we rename all constants that already have a *In* prepended to their name.

Example –

Before	After
I1.*Move(Thing, Place)*: **Action;**	**I2.I1.***Move(Thing, Place)*: **Action;**
I1.*Move(Thing, Place, Thing)*: **Action;**	**I3.I1.***Move(Thing, Place, Thing)*: **Action;**

5.2. Error in library module CARRY

The second error occurred due to conflicting definitions of constant *theme* in module *CARRY*. It stems from the fact that module *MOVE* is *imported* twice by module *CARRY*, once directly and once indirectly through module *GO*. To correct this error, we changed the way *MOVE* is *imported* by *GO*.

Before	After
module GO;	**module** GO;
…	…
import *MOVE*;	**import** *MOVE*;
Thing **is** *Agent*;	*Move*(*x*, *p*) **is exists** *u* (*u*=*x* & *Go*(*u*, *p*));
Move(u, p) **is** *Go(u,p)*;	

Module *CARRY imports* the following definition of constant *theme* from module *GO*

theme(agent, action) : boolean;

It also *imports* the original definition of constant *theme* from module *MOVE*

theme(thing, action) : boolean;

MAD doesn't allow multiple declarations of the same constant (*Move* in this case) with the same number of arguments but different sort types. To overcome this error, we changed the way module *GO imports* module *MOVE*.

5.3. Error in library module CLIMB

There was a type error in the usage of constant *Theme* in an axiom.

Before	**After**
constants	**constants**
Theme(Thing, Action): **Boolean***;*	*Theme(Thing, Action)*: **Boolean**;
axioms	**axioms**
Theme(s, Climb(u, s));	*Theme(x, Climb(u, x))*;
% s is of type Supporter	*% x* is of type *Thing*

In this axiom, we want to say that the *Theme* of action *Climb* is a *Supporter*. However, using a variable (*s*) of type *Supporter* directly in the axiom is incorrect since the first argument of *Theme* should be of type *Thing*. Hence, we change it to a variable of type thing (*x*). The meaning of the axiom remains the same since we declare *Thing* to be a sub-sort of *Supporter*. Therefore, a *thing* (variable *x*) is-a *supporter* (variable *s*), but not the other way round.

5.4. Error in MAD action description

The *Monkey & Bananas* (*M & B*) domain is formalized using three modules. There was a type error in the declaration of a fluent in one of these modules MB. In particular, fluent *TopLevel* should be declared of type “statically determined”, not of type “simple”. This change needs to be made since *TopLevel* is defined as a special case of fluent *Top*, which in turn is defined of type “statically determined” in module *TOP*.

Before	**After**
TopLevel(Thing): **fluent**(*Level*);	*TopLevel(Thing)*: **sdFluent**(*Level*);
…	…
import TOP;	**import** TOP;
Top(x) **is** *TopLevel(x);*	*Top(x)* **is** *TopLevel(x);*

6. Conclusion

This thesis introduced a way to test a general-purpose library of action descriptions and semantics of the MAD language. We translated a complete MAD action description, which used library modules, to the language of the CCalc. The crux of this translation process lay in implementing the semantics of MAD (function δ). We found several bugs in the theoretical description of the library and MAD, and upon correcting these errors, we were able to get correct answers for our queries.

In the future, we plan to continue testing this MAD action description, and other formalizations of toy domain from [Erdogan, unpublished].

7. Acknowledgements

This thesis was greatly influenced by the members of the Texas Action Group (TAG) at Austin and I am grateful for all their help. I would like to thank my supervisor Vladimir Lifschitz for advising this research. It would not have been possible without his guidance.

I would also like to thank Selim Erdogan for helping me implement the translation process (step 2) and all his valuable suggestions throughout my research; Wanwan Ren for helping me find an error in my algorithm; and Yuliya Lierler for suggestions on improving the paper. I would like to thank all of them for their help in finding the errors discussed above. I would also like to thank Greg Plaxton and Bruce Porter for reviewing this thesis.

8. Appendix A – Algorithm code

8.1. Main section

Separate the library modules, and send each to the subroutine *parse_module,* which in turn generates a single-module free of *import* statements.

```
# Read the next line of the file into the variable $line
while ( $line = <LIB> ) {

# If this line contains the reserved word endmodule indicating the end
# of file, then break out of the while loop
  last if $line =~ endmodule;

  # push the line of text just read into an array
  push @module_array,;
}

# send the module just read into the array @module_array to the
# subroutine for processing
my %module = &parse_module(@module_array);
```

8.2. Subroutine parse_module(@array) –

8.2.1. Parsing a non-import sections Parsing an import section -

1. Such a section begins with a line containing one of the keywords except **import.**
2. Read all lines in this section into an array, and then store a pointer to this array in the *hash of arrays*. The end of the section is marked by the beginning of another section i.e. when we come across another keyword.

```
# check if this line contains a keyword
if ( $keywords =~ /($line)/ ) {
  ...
  # reads the next line into $line
  while ( $line = $module[ ++$i ] ) {
    ...
    # checks if this is the end of section
    if ( $keywords =~ /($line)/
         || $line =~ "import"
         || $line =~ "endmodule" ) {
      ...
      # break out of the while loop as we're at the end of section
      last;
    }
    # store this section in an array
    push @array, $module[$i];
  }
  # add this section to the hash of arrays
  $module_hash{$keyword} = [@array];
}
```

8.2.2. Parsing an import section

1. Prepend *I_n* to each occurrence of every variable name, where *n* is the import number.

   Note: A line in the variables section is of the form

```
# Iterate through all the lines in the variables section
foreach my $var_line ( @{ $import_mod_hash{"variables"} } ) {
  ...
  # interate through each of the variables
  foreach my $var (@var_arr) {
    ...
    # Iterate through the axioms section
    foreach my $axiom ( @{ $import_mod_hash{"axioms"} } ) {
      ...
      # prepend I_n to the variable in all axioms it occurs
      $axiom =~ s/(\b$var\b)/I$import_num.$var/g;
    }
    # Prepend I_n to the variable definition
    $var =~ s/^/I$import_num./;
  }
}
```

2. Prepend *I_n* to a constant name if it has been renamed earlier

```
# Iterate through all the lines in the variables section
foreach my $constant ( @{ $import_mod_hash{"constants"} } ) {
  ...
  # Check if this constant has a I_m already prepended i.e. if it has
  # been renamed earlier, then prepend I_n
  if ( $const =~ $inumdot ) {

    # Iterate through the axioms section
    foreach my $axiom ( @{ $import_mod_hash{"axioms"} } ) {

      # prepend I_n to the constant name wherever it occurs
      $axiom =~ s/$const/I$import_num.$const/g;
    }
    # Prepend I_n to the constant definition
    $constant =~ s/$const/I$import_num.$const/;
  }
}
```

3. Check for renaming clauses and handle each case separately

```
while ( $line =~ m/\bis\b/ ) {
```

   a. **Sort renaming** – Replace every occurrence of sort names s_i with s'_i

```
 # Iterate through all the lines in the sorts section
foreach ( @{ $import_mod_hash{"sorts"} } ) {
  # if we find a sort that needs to be renamed
  if (/$var1/) {
    ...
    # Iterate through all the sections
    foreach my $tmp ( @{ $import_mod_hash{$key} } ) {

      # replace s_i i.e. var1 with s_j i.e. var2
      $tmp =~ s/\b$var1\b/$var2/g;
    }
  }
}
```

b. **Constant renaming**

i. Prepend *I_n* to every occurrence of this constant

ii. Insert the equivalences $I\_n.c_j$ ... <-> F_j (j = 1...l) at the beginning of the axioms section.

```
# Insert the equivalences I_n.c_j ... <-> F_j (j = 1...l) at the beginning
# of the axioms section
unshift @{ $import_mod_hash{"axioms"} }, "    $var1 <-> $var2\n";
...
    # Iterate through the axioms and constants sections, as only these
    # can contain legal use of constants
    foreach my $line ( @{ $import_mod_hash{$key} } ) {

      # if this constant was already renamed in the general section
      # in fig ___, then skip it
      next if $line =~ m/I$import_num\.$const/;

      # Prepend I_n to the constant definition
      $line =~ s/\b$const/I$import_num.$const/g;
    }
```

4. Add the processed sections from the imported module and remove duplicates

```
# Iterate through each section
foreach my $key ( keys %import_mod ) {

# Add the contents of the imported module section
  push @{ $module_hash{$key} }, @{ $import_mod{$key} };
}

# Remove duplicate entries in sorts, inclusions and constants sections
foreach my $key (qw/sorts inclusions constants/) {
  ...
  # assign (key => value) pair to line in the section, where the key is
  # the line and value its line number relative to the section
  # by storing in a hash, duplicate entries are automatically removed
  my %section_hash = map { $_ => $i++ } @section_arr;

  # sort the hash by line number
  my @unique_section =
    sort { $section_hash{$a} <=> $section_hash{$b} } keys %section_hash;
```

```
        ...
    }
```

9. Bibliography

Aho, A.V., Kernighan, B. W., Weinberger, P. J., 1988. The AWK Programming Language. *Addison Wesley.*

Arnold, K., Gosling J., and Holmes, D. 2000. The Java Programming Language (3rd Edition). *Addison-Wesley Professional*.

Barker, K.; Porter, B.; and Clark, P. 2001. A library of generic concepts for composing knowledge bases. *In First International Conference on Knowledge Capture*, 14-21.

Clark, P and Porter, B. 1997. Building concept representations from reusable components. *In proceedings of AAAI-97*, pages 369–376.

Dougherty, D. and Robbins, A., 1997. sed & awk (2nd Edition). *O'Reilly Media*.

Erdogan, S., unpublished. A Library of General-purpose Action Descriptions (*Dissertation proposal*).

Erdogan, S.; Ferraris, P.; Lifschitz, V.; Manohar A.; Ren, W. Unpublished. Why the Monkey Needs the Box: A Serious Look at a Toy Domain.

Erdogan, S. and Lifschitz, V. 2006. Actions as special cases. *In Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR).*

Finger, J. 1986. Exploiting Constraints in Design Synthesis. *PhD thesis, Stanford University.*

Geffner, H. 1990. Causal theories for nonmonotonic reasoning. *In Proceedings of National Conference on Artificial Intelligence (AAAI)*, 524–530. AAAI Press.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence* 153(1–2):49–104.

Lifschitz, V. and Ren, W. 2006. A modular action description language. *In Proceedings of the Twenty-First National Conference on Artificial Intelligence.* To appear.

Lin, F. 1995. Embracing causality in specifying the indirect effects of actions. *In Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1985–1991.

McCain, N., and Turner, H. 1997. Causal theories of action and change. *In Proceedings of National Conference on Artificial Intelligence (AAAI)*, 460–465.

McCarthy, J. 1959. Programs with Common Sense. *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*: 75-91.

McCarthy, J. 1979. Ascribing mental qualities to machines. *In Martin Ringle, editor, Philosophical Perspectives in Artificial Intelligence.* Harvester Press, 1979. Reproduced in [McCarthy, 1990].

McCarthy J., 1987. Generality in Artificial Intelligence. *Communications of ACM, 30(12):*1030–1035. Reproduced in [McCarthy, 1990].

Russell, S. and Norvig, P., 2003, Artificial Intelligence: A Modern Approach*. Englewood Cliffs, New Jersey*: Prentice Hall, 2nd edition.

Shanahan, M. 1997. Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia. *MIT Press*.

Stroustrup, B., 2000. The C++ Programming Language (Special 3rd Edition). *Addison-Wesley Professional*.

Wall, L., Christiansen, T., Orwant J., 2000. Programming Perl (3rd Edition). *O'Reilly Media.*