

MAD User's Manual

Selim T. Erdoğan

March 17, 2009

Contents

1	Information about MAD and this manual	3
1.1	Background of MAD and relevant references	3
1.2	Obtaining MAD and System Requirements	3
1.3	About this manual	4
1.4	Contacting Us	4
2	Allowed Input	5
2.1	Comments	5
2.2	Include statements	5
2.3	Identifiers and keywords	6
2.4	Action descriptions	6
2.5	Numeric symbol declarations	6
2.6	Sort declarations	7
2.6.1	Built-in sorts: <code>Boolean</code> , and integer ranges “ $m..n$ ” . . .	7
2.7	Inclusion declarations	7
2.8	Modules	8
2.9	Object declarations	8
2.10	Action constant declarations	9
2.11	Fluent constant declarations	10
2.12	Variable declarations	11
2.13	Axioms	11
2.13.1	Terms	12
2.13.2	Formulas	12
2.13.3	Axioms	14
2.14	Import declarations	16
2.14.1	Sort renaming clauses	16
2.14.2	Constant renaming clauses	16
3	Building and running MAD	19
3.1	Building the code	19
3.2	Running the program	19

4	Using MAD with CCalc	21
4.1	Running CCalc on MAD action descriptions	21
4.2	Issues to watch out for when running MAD with CCalc	22
5	Debugging action descriptions	23
5.1	Common errors in formalizing action domains	23
5.2	Typical mistakes when using library modules	24

Chapter 1

Information about MAD and this manual

1.1 Background of MAD and relevant references

The **M**odular **A**ction **D**escription language MAD was introduced in [Lifschitz and Ren, 2006]. It is a descendant of the $\mathcal{C}+$ action language described in [Giunchiglia *et al.*, 2004] and of the “the Causal Calculator” (CCalc)¹. Some motivation and theoretical background for extending $\mathcal{C}+$ with library modules can be found in [Erdoğan and Lifschitz, 2006] and an early example of MAD being used can be found in [Erdoğan *et al.*, 2007]. Recent work in [Lifschitz and Ren, 2007] has begun to extend the semantics of $\mathcal{C}+$, which will also be inherited by MAD.

1.2 Obtaining MAD and System Requirements

MAD can be downloaded from its homepage:

`http://www.cs.utexas.edu/~tag/mad/`

The software is written in the C programming language. In order to build MAD from source, one needs a C compiler, a Lex-like lexical analyzer generator, and a Yacc-like parser generator.

We have tested our software on Unix/Linux-like systems and used the following programs to build MAD from source:

- GCC, the GNU Compiler Collection
- flex, The Fast Lexical Analyzer
- bison, the GNU parser generator

¹`http://www.cs.utexas.edu/tag/cc/`

More specifically, we have successfully tested MAD with the following combinations:

- Debian GNU/Linux (sid), gcc 4.3.3, flex 2.5.35, bison 2.4.1
- SunOS 5.9, gcc 4.2.2, flex 2.5.33, bison 2.0

1.3 About this manual

This manual describes MAD version 0.4.

Things written in **typewriter font** indicate input that should appear exactly as shown. Things written in *italics* refer to an element of input which has already been explained or is about to be explained.

1.4 Contacting Us

Please send any questions/comments/bug reports to `selim@cs.utexas.edu`.

Chapter 2

Allowed Input

A MAD input file contains an action description, along with comments.

2.1 Comments

A comment in the input begins with the character `%` and lasts until the end of the line (i.e. until a newline character). Everything in the comment is ignored by the parser.

2.2 Include statements

An input file may contain one or more “include” statements before the action description. These statements specify (within quotes) names of other files containing action descriptions. They are treated as if the contents of the included files appear in place of the include statements. For example, to include a file named `library` (located in the same directory) before an action description, we can write the following:

```
include "library"  
action_description_which_may_refer_to_modules_from_included_file
```

If the filename specifies just a simple filename, the file is expected to be in the directory in which the program is executed. If a relative path to a file is given, this is interpreted as relative to the directory in which the program is executed.

If the included file does not contain a complete action description, an error will occur.

Include statements can only appear at the start of a file, and the final include statement in a file must be followed by an action description.

Include statements may be nested in the sense that a file which is referred to in an include statement may contain an include statement itself.

2.3 Identifiers and keywords

Identifiers must begin with a letter. The letter may be followed by a combination of letters, numbers, or “_”. In `grep`-like regular expression notation, we may write this as

Identifier: `[a-zA-Z][a-zA-Z0-9_]*`

(Note for those familiar with the semantics of MAD: this notation allows *Integer* to appear in an identifier, though any appearance of a “.” will cause a syntax error.)

The following words are keywords of the language and may not appear as identifiers:

`include, module, import, is, case, numeric_symbol, sorts, inclusions, objects, actions, fluents, variables, action, explicitAction, simple, staticallyDetermined, rigid, axioms, if, after, constraint, default, exogenous, causes, nonexecutable, always, may, cause, inertial, exists, forall, true, false, Boolean`

2.4 Action descriptions

An action description is composed of a series of four basic components:

- numeric symbol declarations
- sort declarations,
- inclusion declarations,
- modules,

which may be interleaved. These components are described below.

2.5 Numeric symbol declarations

A sort declaration is of the form

`numeric_symbol id=int`

where *id* is an identifier and *int* is an integer.

The purpose of such declarations is to have symbolic names for integers. Any occurrence of *id* occurring after the above numeric symbol declaration will be treated as an occurrence of *int*.

Identifiers declared as numeric symbols may only occur in places where integers are allowed.

2.6 Sort declarations

A sort declaration is of the form

```
sorts
s1 ;
⋮
sn ;
```

where $1 \leq n$, and each s_i is an identifier or the keyword `Boolean`.

A sort may not be declared more than once.

2.6.1 Built-in sorts: Boolean, and integer ranges “ $m..n$ ”

The keyword `Boolean` is a built-in sort name, declared implicitly. (Later, in the section about objects, we will see that this built-in sort contains two built-in objects, `true` and `false`.)

An expression of the form “ $m..n$ ”, where m and n are integers, represents the set of integers between m and n inclusively. Such integer ranges are built-in sorts in MAD (with the integers as built-in objects belonging to those sorts) and they are considered to be declared implicitly.

Declaring `Boolean` is allowed but a warning is given. Declaring integer ranges as sorts is prohibited.

Definition: We call the keyword `Boolean`, integer ranges of the form “ $m..n$ ”, and sort names s_i declared as above *simple sort names*.

2.7 Inclusion declarations

An inclusion declaration is of the form

```
inclusions
i1 ;
⋮
in ;
```

where $0 \leq n$, and each i_i is an inclusion expression, as defined below.

An inclusion expression is of the form

$$s_1 \ll s_2 \ll \dots \ll s_n$$

where ($n > 1$), s_1 is a simple sort name other than an integer range, and s_2, \dots, s_n are sort names other than `Boolean`.

The reason that the keyword `Boolean` or an integer range cannot appear as a supersort (on the right of \ll) is that such an appearance would lead to these

built-in sorts having objects other than their built-in objects.

We may think of the sort and inclusion declarations as forming a forest, with the sorts as vertices and each \ll relation standing for an edge from the right-hand sort to the left-hand sort.

Inclusions leading to cycles in the sort inclusion forest are prohibited. (Attempting to make such declarations will cause the system to give an error message.)

2.8 Modules

A MAD module is of the form

```
module module-name ;
```

```
module-body
```

where *module-name* is an identifier. The *module-body* consists of an ordered series of sections, any of which may appear at most once. These are, in order, object declarations, action constant declarations, fluent constant declarations, variable declarations, and axioms. In addition, the *module-body* may contain any number of import declarations in before or after (but not within) these sections.

Each module in an action description must have a unique name.

2.9 Object declarations

An object declaration is of the form

```
objects  
o-spec1 ;  
:  
o-specn ;
```

where $0 \leq n$, and each *o-spec*_{*i*} is an object specification, as defined below.

An object specification is of the form

```
o1, ..., om : sort
```

where $1 \leq m$, each *o*_{*i*} is an identifier, possibly followed by a parenthesized list of arguments, and *sort* is a sort name other than **Boolean**. Any arguments following *o*_{*i*} must be a simple sort name.

Example. Here is an example object declaration, assuming sorts **Person**, and **Object** have been declared:

```

objects
Player(1..10) : Person;
Hat(Person) : Object;

```

The keywords `true` and `false` are built-in objects, of sort `Boolean`, declared implicitly in every module. These objects cannot be declared explicitly. Also, no other objects may be declared to be of sort `Boolean`.

Integers are also built-in objects, and every integer is considered to be declared implicitly, so that it can be used in formulas, as will be seen below.

We may think of the sort declarations, inclusion declarations, and object declarations as forming a “sort dependency” forest. The vertices are sorts. Each object o_i specified above adds an edge for every argument it has. The edge points from *sort* to the argument. Inclusions of the form “ $s_1 \ll s_2$ ” add edges from s_2 to all of the sorts which s_1 already points to.

Example. The example object declaration above adds the edges

$$\text{Person} \rightarrow 1..10$$

$$\text{Object} \rightarrow \text{Person}$$

to the sort dependency forest.

Cycles in the sort dependency forest are prohibited. (Attempting to make any declarations which result in a cycle will cause the system to give an error message.)

An object may not be declared more than once, except when the re-declaration is part of an imported module. (See the section on import declarations below.) The re-declaration must match the original declaration, i.e., the two declarations must assign the same sort to the object.

2.10 Action constant declarations

An action constant declaration is of the form

```

actions
c1 ;
⋮
cn ;

```

where $1 \leq n$, and each c_i is an identifier, possibly followed by a parenthesized list of arguments. An action constant argument is a simple sort name. No domain is specified when declaring an action because all actions are `Boolean`-valued.

Example. Here is an example action constant declaration, assuming sorts `row`, `column`, and `color` have been declared:

```
actions
Paint_square(row, column, color);
```

The arguments of an action constant being declared may not contain the keyword **action**. (Otherwise, imagine two actions $a(\mathbf{action})$ and b . When we ground, we would get b , $a(b)$, $a(a(b))$, etc., never ending.)

An action constant may not be declared more than once, except when the re-declaration is part of an imported module. (See the section on import declarations below.) The re-declaration must match the original declaration, i.e., the two declarations must assign the same arguments.

2.11 Fluent constant declarations

A fluent constant declaration is of the form

```
fluents
c_spec1 ;
:
c_specn ;
```

where $1 \leq n$, and each c_spec_i is a fluent constant specification, as defined below.

A fluent constant specification is of the form

$$c_1, \dots, c_m : \mathit{kind}(\mathit{domain})$$

where $1 \leq m$, and each c_i is an identifier, possibly followed by a parenthesized list of arguments. A fluent constant argument is a simple sort name or the keyword **action**. The *kind* may be one of the keywords **simple**, **staticallyDetermined**, or **rigid**, indicating, respectively, a simple fluent, statically determined fluent or rigid fluent. The *domain* is a simple sort name.

When declaring simple fluents, statically determined fluents or rigid constants, it is also possible to omit the parenthesized domain, in which case it is implicitly assumed to be (**Boolean**).

Example. Here are some example fluent constant declarations assuming sorts **row**, **column**, and **color** have been declared:

```
fluents
Game_started: rigid;
Square_color(row, column) : simple(color);
```

A fluent constant may not be declared more than once, except when the re-declaration is part of an imported module. (See the section on import declarations below.) The re-declaration must match the original declaration, i.e., the two declarations must assign the same arguments, the same kind and the same domain to the fluent constant.

2.12 Variable declarations

A variable declaration is of the form

```
variables
v_spec1 ;
⋮
v_specn ;
```

where $0 \leq n$, and each v_spec_i is a variable specification, as defined below.

A variable specification is of the form

$$v_1, \dots, v_m : sort$$

where $1 \leq m$, each v_i is an identifier, and *sort* is a simple sort name, the keyword **action** or the keyword **explicitAction**.

A variable may not be declared more than once in the same module.

Each variable is local to the module in which it is declared. To use the same identifier as a variable in another module, one has to declare it anew in the latter module (and may declare it as belonging to a different sort).

2.13 Axioms

An axioms section is of the form

```
axioms
axiom1 ;
⋮
axiomn ;
```

where $0 \leq n$, and each $axiom_i$ is an axiom, as defined below.

The axiom section of a MAD action description is different from the declaration sections we described above, in that the previous sections serve to *declare* identifiers whereas this section *uses* these identifiers. From now on, whenever we refer to a constant/object/variable, we mean an identifier which was declared as such in a prior section of the action description.

Axioms in MAD are like causal laws in $\mathcal{C}+$. They are composed of *formulas* and certain keywords such as **caused**, **if**, **after**, etc. We will show the exact

forms of acceptable axioms below, but first we need to define what we mean by a valid formula. Formulas are defined recursively, using *terms* and logical connectives. Therefore, to describe what a valid formula is, we need to first define what a term is.

2.13.1 Terms

A *term* may be

- a constant (followed by a parenthesized list of arguments if so declared)
- a variable
- an object (followed by a parenthesized list of arguments if so declared)
- an integer
- (*term*)
- *term* + *term*
- *term* * *term*

The parenthesized list of arguments following a constant/object must match the arguments listed in the declaration of the constant/object, i.e. the arguments in the constant/object term must be objects/variables of the sort declared for the corresponding argument, or a action constant if the corresponding argument was declared as an action.

Since a constant declaration allows only simple sort names or **action** as arguments, and object declarations only allow simple sort names as arguments, we would normally expect no non-action constants to be allowed as arguments of a constant/object term. However, the system allows such arguments as shorthand for the value the constant has (at that time instant). In other words, an argument for a constant/object term may be a constant term, provided that the latter's domain matches the sort of the former's argument declaration. Formulas with such shorthand notation are expanded to a certain longhand form. The exact details will be given after formulas are defined.

The last two items above, expressions for arithmetic operations (addition and multiplication) may not appear as arguments to constants/objects. Furthermore, each of the terms in these expressions must be “numerical,” meaning an integer, a constant with an integer range domain, a variable with an integer range sort, or another validly formed arithmetic expression.

2.13.2 Formulas

A formula is built from terms and connectives in one of the following ways:

1. zero-place connective **true**
2. zero-place connective **false**

3. a Boolean constant term
4. an action variable
5. *sort-name* (*variable*)
6. **Boolean** (*variable*)
7. **action** (*variable*)
8. *term* = *term*
9. *term* != *term*
10. *term* < *term*
11. (*formula*)
12. *-formula*
13. *formula* & *formula*
14. *formula* | *formula*
15. *formula* -> *formula*
16. *formula* <-> *formula*
17. **exists** *variable formula*
18. **forall** *variable formula*

Items (12) through (16) correspond to the usual logical connectives: (in descending order of precedence) negation, conjunction, disjunction, implication and equivalence. All of the binary connectives are left-associative.

Quantifiers **exists** and **forall** have lower precedence than the logical connectives, so, for example, assuming Q_1, Q_2 are quantifiers, v_1, v_2 are variables, and F, G are formulas,

$$Q_1 v_1 F \text{ connective } Q_2 v_2 G$$

will be parsed as

$$Q_1 v_1 (F \text{ connective } Q_2 v_2 G)$$

and not as

$$(Q_1 v_1 F) \text{ connective } (Q_2 v_2 G)$$

Items (5) is shorthand for a quantified formula

$$\mathbf{exists} \text{ new-variable-of-sort } \textit{variable} = \text{new-variable-of-sort}$$

and is called a “sort name formula.”¹ Items (6) and (7) are similar — they are sort name formulas for built-in sort **Boolean** and for actions.

In items (8) and (9), there are some additional constraints that are checked:

¹A sort name formula like (5) corresponds to the English sentence “*variable* is of sort *sort-name*.”

- if one of the terms is an action variable or an explicit action variable, then the other term must be an action variable or an explicit action variable or a Boolean action constant.
- if both of the terms are constants, they must have the same domain
- if one of the terms is a constant and the other is an object/variable, the sort of the object/variable must be the same as the domain of the constant.

In item (10), each term must be numerical, meaning an integer, a constant with an integer range domain, a variable with an integer range sort, or an arithmetic operator term as given in the definition of a term (addition and multiplication).

As mentioned above in the description of terms, MAD also allows having constants as arguments to constant terms, even where a non-action sort is expected. This is shorthand for the value of that constant at the time. For example,

`Square_color(row_of(queen), column)`

is valid shorthand if the domain of `row_of` matches the first argument declaration of `Square_color`. This would be expanded by using a new variable, say `r`, from the domain of `row_of`. The minimal formula F in which this term appears would be expanded to

$$F' \ \& \ \text{row_of}(\text{queen})=r$$

where F' is obtained by replacing that occurrence of `row_of(queen)` by `r` in F . Note that such shorthands may be nested too.

IMPORTANT: This shorthand is only allowed in formulas appearing as parts of axioms, not in terms appearing as parts of import declarations. (We will describe import declarations below.)

2.13.3 Axioms

An axiom is built from formulas and terms in one of the following ways: (The parts within square brackets are optional)

1. *formula* [*if formula*] [*after formula*]
2. *formula causes formula* [*if formula*]
3. *default formula* [*if formula*] [*after formula*]
4. *exogenous constant* [*if formula*]
5. *inertial fluent-constant* [*if formula*]
6. *constraint formula* [*after formula*]
7. *nonexecutable formula* [*if formula*]
8. *always formula*

9. *rigid fluent-constant*

10. *formula may cause formula* [if *formula*]

Recall, from the paper “Nonmonotonic Causal Theories” [Giunchiglia *et al.*, 2004], that

- an *action formula* is a formula with no fluent constants and at least one action constant,
- a *fluent formula* is a formula with no action constants.

In items (2) and (10) the first formula has to be an action formula and the second should be an action formula or a fluent formula.

In item (7) the first formula has to be an action formula.

All of the axioms listed above may be seen as special cases of item (1).² Let us rewrite this general form as

$$F \text{ [if } G \text{] [after } H \text{]}$$

where F, G and H are formulas. An axiom of this form must satisfy the following conditions in order to be valid:

- F must be such that the axiom is definite, i.e. F is either
 - the zero-place connective false
 - a single atomic formula³ with at most one constant
 - the negation of an atomic formula with exactly one Boolean constant
- if there is no H part and F is a fluent formula, then G must be a fluent formula
- if there is an H part,
 - F and G must be fluent formulas
 - F must not contain any statically determined fluents or rigid constants.
- if F contains a rigid constant, the axiom must not contain any non-rigid constants.

The axioms listed in this section cover all of the abbreviations from Appendix B of [Giunchiglia *et al.*, 2004], with the exception of abbreviations (15)-(17) there, which involve the **unless** construct.

²The details of how items (2)-(10) may be seen as abbreviations of (1) can be found in Appendix B of [Giunchiglia *et al.*, 2004].

³Items (1)-(4) and (7) in the description of formulas above.

2.14 Import declarations

An import declaration is of the form

```
import module-name ;  
sort-renaming-clause1 ;  
⋮  
sort-renaming-clausen ;  
constant-renaming-clause1 ;  
⋮  
constant-renaming-clausem ;  
where each  $(n, m \geq 0)$ .
```

An import declarations section of a MAD action description is similar to the axioms section in that all of the identifiers used must have been declared prior to this section. On the other hand, an import section *implicitly declares* any identifiers from the module it imports.

In the following sections on sort and constant renaming clauses, we will refer to the module being imported as M_1 and the module importing it as M_2 .

2.14.1 Sort renaming clauses

A sort renaming clause is of the form

s_1 is s_2

where s_1 is a sort which has been declared prior to M_1 and s_2 is a sort declared prior to M_2 . Neither of these two sorts may be **Boolean** or an integer range.

(If s_2 were **Boolean** or an integer range, then any objects declared to be of sort s_1 in M_1 would become objects of this built-in sort which has all of its objects predefined.)

A sort may not be renamed more than once in the same import. (i.e. it may not appear on the left hand side of more than one sort renaming clause.)

2.14.2 Constant renaming clauses

There are two kinds of constant renaming clauses, depending on whether the constant being renamed is Boolean-valued or not.

Boolean constant renaming clauses

If the constant being renamed is Boolean-valued, a renaming clause is of the form

$c(v_1, \dots, v_n)$ is *boolean-const-renaming-rhs*₁

or

$c(v_1, \dots, v_n)$ is
case *formula*₁ : *boolean-const-renaming-rhs*₁ ;

```

:
case formulak : boolean-const-renaming-rhsk ;
[default : boolean-const-renaming-rhsk+1 ;]

```

where c is a constant of M_1 , v_1, \dots, v_n are variables/objects of M_2 or integers, $n \geq 0$, $k \geq 1$. Each variable/object must be of the sort declared for the corresponding argument, or a subsort of that sort, and all variables must occur at most once. Any object appearing as an argument must be “fully instantiated”, meaning it cannot have any variable arguments itself. Each $formula_i$ must have no constants and should not contain any variables other than v_1, \dots, v_n . The part within square brackets is optional. If not included, it is filled in to have *boolean-const-renaming-rhs_{k+1}* be **false**.

In this clause *boolean-const-renaming-rhs_i* is one of

- a Boolean constant term
- *a-Boolean-constant-term* = **true**
- *a-Boolean-constant-term* = **false**
- **true** = *a-Boolean-constant-term*
- **false** = *a-Boolean-constant-term*
- *-a-Boolean-constant-term*
- **true**
- **false**

The constant term may not have any other constants appearing as shorthand. (i.e., when viewed as a formula, this term must not be expandable to one in which more than one constant occurs.)

Nonboolean constant renaming clauses

If the constant being renamed is not Boolean-valued, a renaming clause is of the form

$c(v_1, \dots, v_n)$ is *nonboolean-const-renaming-rhs₁*

or

```

c(v1, ..., vn) is
case formula1 : nonboolean-const-renaming-rhs1 ;
:
case formulak : nonboolean-const-renaming-rhsk ;
default : nonboolean-const-renaming-rhsk+1 ;

```

where c is a constant of M_1 , v_1, \dots, v_n are variables of M_2 , $n \geq 0$, $k \geq 1$. Each variable must be of the sort declared for the corresponding argument (It cannot be a variable of a subsort of the argument declaration), and all variables must occur at most once. Each $formula_i$ must have no constants and should not contain any variables other than v_1, \dots, v_n .

In this clause *nonboolean-const-renaming-rhs_i* is one of

- a constant term with the same domain as c
- an object which belongs to the domain of c

Additional constraints on constant renaming clauses

There are also some additional constraints depending on the kind of constant appearing on the left hand side of the **is** keyword:

- if it's an action constant, any constant on the right hand side of **is** must be an action constant.
- if it's a statically determined fluent constant, any constant on the right hand side of **is** must be a fluent constant.
- if it's a simple fluent constant, any constant on the right hand side of **is** must be a simple fluent constant.
- if it's a rigid constant, any constant on the right hand side of **is** must be a rigid constant.

The same constant may not be renamed more than once in the same import declaration.

Chapter 3

Building and running MAD

3.1 Building the code

As mentioned in the first chapter, we used the tools `gcc`, `flex` and `bison` on Unix/Linux systems.¹ Once you have these installed, to build the program from source, simply run the command `make` at the shell prompt. This will produce an executable named `mparse`.

3.2 Running the program

`mparse` is a program which parses its MAD input and processes the import statements, building internal data structures corresponding to each module in the input, finally printing these processed modules to an output file.

The programs expects to be called as follows:

```
mparse [-i] input_file_1 input_file_2 ... input_file_n [-o output_file]
```

If no output file is specified, the output is written to a file named `mparse.output`.

When more than one input file is specified, `mparse` treats them as if they were all concatenated in one big file. The optional switches `-i` and `-o` may occur in any place in the command line, not just at the beginning or the end. However, the `-o` switch must be followed by an output file.

By default, `mparse` automatically generates the CCalc input file corresponding to the final module in the MAD action description.

IMPORTANT: CCalc doesn't allow identifiers to have free capitalization, so the output of `mparse` will turn all letters in identifiers into their lower-case form. The program will print a warning message if two identifiers become the same when converted to lower-case.

¹If you want to replace `gcc`, `flex`, or `bison` with other software that accomplishes the same job, you will need to change the makefile.

If the program is called with the `-i` (interactive) option, then after parsing, instead of directly generating CCalc code, the user is asked to choose among different options to view the resulting modules:

1. Print modules parsed: prints the data structures built for all of the modules, with the import statements processed.
2. Print last module: prints only the data structures built for the last module in the MAD description, with the axioms ground.
3. Print last module as CCalc input: like the preceding option, but prints the module in CCalc format.
4. Quit without printing anything.

Chapter 4

Using MAD with CCalc

4.1 Running CCalc on MAD action descriptions

We don't explain the details of running CCalc in this manual. For that, the reader is referred to the ccalc homepage:

<http://www.cs.utexas.edu/~tag/cc/>

As explained in the preceding chapter, the MAD executable `mparse` can turn a given MAD action description into a CCalc input file, albeit with the identifiers turned into all-lower-case words. Therefore, when writing a CCalc query to be used with the given file, one must write all identifiers as all-lower-case.

In the course of developing an action description, it is often the case that a user first runs `mparse` on a MAD description, prepares a set of CCalc queries, loads both of them into CCalc and tests them, only to find that he has to go back to make some changes to the MAD description, and tests with the same queries. In order not to change any files other than the MAD input file, we recommend using the CCalc include statement

```
:- include 'ccalc-input-filename'.
```

at the beginning of files containing queries. This is the approach we have taken in the examples provided as part of the MAD distribution. Each CCalc query file, (e.g. `bw-queries`) begins with a CCalc include statement specifying the name of the file we expect `mparse` will generate (e.g. `bw.cc`). Then, when running `mparse` we use the `-o` option to tell it to generate an output file with this name (e.g. `mparse bw -o bw.cc`).

4.2 Issues to watch out for when running MAD with CCalc

- The CCalc file generated by `mparse` includes a `show` specification which hides certain constants from the output. Those hidden are the renamed constants beginning with an import prefix of the sort “*Integer.*” and also the constants “**Actor**” and “**Theme**” which are declared in the MAD library. (These will be hidden even if the user declares them himself, without reference to the library.)
- Make sure that you don’t have similar identifiers with different capitalization. In MAD, `P2` and `p2` are different identifiers because MAD is case-sensitive. But during the automatic conversion of MAD files into CCalc input, they’ll both turn into `p2`. The MAD executable `mparse` will give a warning in such cases but will still print out a CCalc input file. (If one uses such a problematic file with CCalc, such errors are very hard to figure out because CCalc won’t complain even if the same identifier is declared as both an object and a variable.)

Chapter 5

Debugging action descriptions

5.1 Common errors in formalizing action domains

Often the first attempt to formalize an action domain is not successful and running CCalc yields no solution. In such cases the first thing to do is to run two simple queries to check whether the transition system corresponding to the description has any valid states and transitions.

```
% Tests whether the transition system has any valid states
:- query
maxstep :: 0;
label :: 0.
```

```
% Tests whether the transition system has any valid transitions
:- query
maxstep :: 1;
label :: 1.
```

If the first query succeeds but the second fails, this means that even though there are causally explained states in the transition system, no causally explained transition exists. This is most often because an action instance has no cause specifying whether it should be executed or not. Usually we make action constants exogenous and don't specify any further cause for them. So a common mistake is to forget to make an action constant exogenous.

A related mistake occurs when we import another module and define a new action in terms of one from the imported module. If the old action was declared to be exogenous we usually won't specify that the new one is exogenous too, because that follows from the two actions being equivalent. However, if the equivalence does not cover all instances of the new action, then a cause must be explicitly specified for these other instances. For example, the following action

description snippet declares a new sort **Container** which is a subsort of the library sort **Thing**, and a new action **PutIn** is defined in terms of library action **Mount**.

```
inclusions
  Container << Thing;

constants
  PutIn(Thing, Container) : action;

variables
  x : Thing;
  c : Container;

import MOUNT;
Mount(x,c) is
  case -Agent(x) : PutIn(x,c);
```

Here the import statement specifies that the two actions are equivalent only when their first arguments are not of library sort **Agent**. (This latter sort is a subsort of **Thing**.) So, even though action **Mount** is exogenous, instances of **PutIn** where the first argument is an **Agent** don't inherit this property.

5.2 Typical mistakes when using library modules

- Using module **LOCAL** for actions which don't have any **Actor** or **Theme** defined.
- Forgetting to include **TOP** for domains which have both a concept of support and a concept of locations involved. Forgetting to define a **TopLocation** value for all **Things**.

Bibliography

- [Erdoğan and Lifschitz, 2006] Selim T. Erdoğan and Vladimir Lifschitz. Actions as special cases. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 377–387, 2006.
- [Erdoğan *et al.*, 2007] Selim T. Erdoğan, Paolo Ferraris, Vladimir Lifschitz, and Wanwan Ren. Why the monkey needs the box: A serious look at a toy domain¹ In *Working Notes of the Seventh IJCAI International Workshop on Nonmonotonic Reasoning, Action and Change*, 2007.
- [Giunchiglia *et al.*, 2004] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.
- [Lifschitz and Ren, 2006] Vladimir Lifschitz and Wanwan Ren. A modular action description language. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 853–859, 2006.
- [Lifschitz and Ren, 2007] Vladimir Lifschitz and Wanwan Ren. The semantics of variables in action descriptions. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, 2007.

¹<http://www.cs.utexas.edu/~vl/papers/serious.ps> .