Logitest
User's Guide

Marta Cialdea Mayer

January 2009

1 Functionalities of the system

Logitest is a simple support system for the design and check of test items of the form "given the information T, which of the following assertions is derivable/underivable/consistent/inconsistent with T?".

The system takes a file in input, specifying the components of the item and the tasks to be executed. The program is called from a command shell (both under Linux and Windows) with the syntax:

```
logitest <filename>
```

where `filename` is the name of the file containing the specification of the problem. The system outputs its results both on the screen and on the file `<filename>.out`

The input file contains a description of the language used in the test (predicates, object types, constants, ecc.) and the item stem (the information T), represented by a set of logical formulae. Moreover it contains (some or all of) the following specifications:

- a set of formulae that are to be tested for derivability from T;
- a set of formulae that are to be tested for consistency with T;
- a set of formulae representing a plausible incorrect understanding of T (students' possible misconceptions, that will be called "*beliefs*" in the sequel), that are used to generate plausible distractors;
- directives asking the system to generate different components of the item.

The syntax used in the input file is first-order, but its components are translated into a propositional language. In fact, the program can only deal with questions involving a finite and fixed set of objects (which are assumed to be pairwise different).

The tasks that can be accomplished by the system are described below.

1.1 Derivability and consistency check

If the input file specifies a set of formulae to be checked for derivability from T or consistency with T, the system checks them one by one. This is the basic task, useful to check whether one and only one of the options represents the correct answer.

1.2 Completion of the information given in the item stem

This task is useful when the designer has run out of ideas; in this case the item stem specified in the input file contains general information and possibly some facts, that however are not sufficient to derive some given conclusion C, representing the correct option. The system generates all the minimal conjunctions of literals E_i, that are consistent with T and such that $T \cup \{E_i\} \models C$ (excluding trivial ones, i.e. such that $E_i \models C$). In other terms, the system solves the abduction problem given by T and C. A set of *abducible predicates* can also be specified, in order to obtain solutions containing only such predicates.

Obviously, the item stem can be completed by addition of (a formula equivalent to) the disjunction of two or more solutions.

1.3 Generation of correct answers

The system can be asked to generate formulae which are derivable from the item stem. Logical consequences of the stem are minimal disjunctions of literals, excluding trivial ones (that are already explicitly present in the item stem). If abducible predicates have been defined, only formulae built out of them are generated.

The system can also generate literals that are consistent/inconsistent with the stem (excluding its logical consequences). More complex consistent/inconsistent formulae can be obtained from the literals by use of logical operators.

1.4 Generation of distractors

The specification file may include a belief set, that is intended to represent possible students' *misconceptions* or an incorrect interpretation of the stem (such as, for instance, reading a logical equivalence in place of an implication). The *beliefs* are used to generate incorrect though "plausible" options. The system generates minimal disjunctions C of literals such that $B \models C$ and $T \not\models C$, where B is the belief set and T the item stem. Also in this cases, only abducible predicates are used, if they have been declared.

2 Syntax of the input file

In what follows, `<ident>` is an *identifier*, i.e. a string beginning with an alphabetic character, possibly followed by alphanumeric characters and the special character _ (underscore).

1. Comments:

   between `/*` and `*/`

   from `;` or `#` to the end of the line

2. Language definition. The first declarations in the input file must define: a set of types, a set of predicates (each one with the associated arguments' types), a set of constants to denote individuals of each type. Optionally, a set of abducible predicates can be declared. The syntax for such declarations is the following (carriage returns are ignored):

```
Types: <type> <type> ... <type>.
Predicates: <pred>(<type>,...,<type>)
              ...
             <pred>(<type>,...,<type>).
Objects: <ident> ... <ident> : <type> -
         <ident> ... <ident> : <type> -
           ..........
         <ident> ... <ident> : <type> .
Abducibles: <pred> ... <pred>.
```

Here, `<type>` and `<pred>` are identifiers used to name types and predicates, respectively.

The declaration of a predicate (`<pred>(<type>,...,<type>)`) is made of the predicate, followed by the types of its arguments, enclosed in parenthesis and separated by commas. If a predicate can be applied to different types of arguments, it must be declared twice (or more times). For instance, if `person, house, country` are types, we could declare:

```
Predicates: lives(person,house) lives(person,country).
```

3. Declaration of the item stem (*theory*) and the belief set.

```
    Theory: <form> , ... , <form>.
    Beliefs: <form> , ... , <form>.
```

Here, `form` is a formula (see item 5).

The declaration of the beliefs is optional. When it is included, the system automatically generates disjunctions of literals derivable from the beliefs.

4. Task declaration. Abduction, deduction and consistency checks, and generation of logical consequence of the theory and literals consistent or inconsistent with the theory are asked for by the following directives:

```
Abduce: <form>.
Deduce: <form>,...,<form>. /* deducibility checks */
Consistent: <form>,...,<form>. /* consistency check */
Generate: Derivable Consistent Inconsistent.
```

The keyword `Generate` can be followed by any (non-empty) sequence of the keywords `Derivable Consistent Inconsistent` (in any order).

5. Formulae.

Constants can be either object names or integers. A variable (`var`) is any identifier that is neither a constant, nor the name of a type or predicate. Terms (`<term>`) are either constants or variables.

The syntax for formulae is the following:

```
<ident> /* propositional atom */
True
```

```
False
<ident>(<term>,...,<term>)
<term> = <term>
<form> <-> <form>
<form> -> <form>
<form> & <form>
/* or, equivalently */ <form> and <form>
<form> | <form>
/* or, equivalently */ <form> or <form>
- <form>
/* or, equivalently */ not <form>
/* or */  ~ <form>
forall <var>...<var> : <type> <form>
exists  <var>...<var> : <type> <form>
/* or, equivalently */   forsome  <var>...<var> : <type> <form>
( <form> )
```