6: Implementation of SPASS with Axiomatic Translation of Modal Logic.

DRAFT

Extensions to SPASS were implemented in C. Several general principles were applied to design of the code. Where possible all new functions of the eml-module are labeled as `static`. In order to prevent cyclic lists of `TERM`s (which crash SPASS), `TERM`s were copied to form new instances of `TERM` wherever sensible (using `term_Copy()`). Functionality is broken-up into small and coherent functions, which are organized by a series of controlling functions. A better design could have been arrived at in an object-oriented language, but this facility this is not available in C. A test system in Java was developed with an object oriented design (see section 6.13). The code referred to in the text is available from the project web site. A list of pre-existing SPASS functions that were used in the eml-module is given in the Appendix. Instruction for dwnload, compilation, etc are given on the project web site, and reproduced in the Appendix.

6.1. Data Structures:

Several data structures were introduced into the eml-module in order to support the axiomatic translation. These are used to store parameters (both taken directly from the input, and derived) related to the translation.

6.1.1 Axioms:

The parameters related to axioms are stored in a `struct` of type `AXIOM`. The elements of this structure are

- `AXIOMTYPE AxiomType` – taking values `K, FOUR, FIVE, T, D, B, ALT1, FOURK, FIVEK, ALT1KK AXIOM_NONE, SR, G, DEN, TR, M, W, CR, CR2, CR3, BK, H, DBBB` - defining the name of the axiom.
- `AXIOMMODE Mode` – taking values `AXIOMATIC_TRANSLATION, AXIOMATIC_TRANSLATION_WITH_COMPOSITION, CLASSICAL_TRANSLATION` - defining the type of translation applied.
- `int Parameter1` – defines the factor κ for axiom 5^{κ} and 4^{κ}, and factor κ_1 for axiom $alt_1^{\kappa 1 \kappa 2}$.
- `int Parameter2` – defines the factor κ_2 for axiom $alt_1^{\kappa 1 \kappa 2}$.
- `int Index` – records the order in which the axioms are applied during translation
- `SYMBOL Modality` – the modality symbol - only important for multi-modal input formulae.
- `SYMBOL Modality2` – the second modality symbol - only important for bimodal axioms.

The `AXIOM`s to be applied are accumulated in the array `AxiomComposition[]`. This is referred to later as the *axiom cache*. This array can potentially hold `MAXDEFINEDAXIOMS(100)` elements, and is available to any function in the eml-module. The array is populated in the order in which axioms are applied during translation, and the `int AxiomCompositionIndex` points to the next empty slot in this array.

There are utility functions associated with AXIOMs. New axioms are created by a group of related functions. `eml_BuildAxiomRS()` builds an axioms at the next empty slot in `AxiomComposition[]`, and has formal parameters corresponding to each of the elements listed above. In `eml_BuildAxiom()` and `eml_BuildAxiomR()` the modalities are assigned a default value of `symbol__NULL`. `eml_BuildAxiomAt()` populates a specific slot in `AxiomComposition[]`. The `AxiomComposition[]` array may be accessed by `eml_GetAxiom()`. If `AxiomCompositionIndex` is greater than 1, then `TRUE` is returned from `eml_AreAxiomsDefined()`.

Note: in these tables parameter names are only included where they reduce ambiguity.

Table 6.1. Functions Manipulating Axioms:

`BOOL` **`eml_BuildAxiom`**`(AXIOMTYPE,AXIOMMODE, int Parameter1, int Parameter2)`
`BOOL` **`eml_BuildAxiomR`**`(AXIOMTYPE,AXIOMMODE,int Parameter1,int Parameter2,SYMBOL)`
`BOOL` **`eml_BuildAxiomRS`**`(AXIOMTYPE,AXIOMMODE,int Parameter1,int Parameter2,SYMBOL s1,SYMBOL s2)`
`BOOL` **`eml_BuildAxiomAt`**`(int Index,AXIOMTYPE,AXIOMMODE,int Parameter1,int Parameter2)`
`AXIOM` **`eml_GetAxiom`** `(int Index)`
`BOOL` **`eml_AreAxiomsDefined`**`()`

6.1.2 Propositional Variables:

Once a propositional variable has been created, then it is stored, for easy reuse, in a `SYMBOL` array (of size `eml_SIZEAXIOMVARSTORE,100`) called `eml_AxiomVarStore[]`. This is referred to as the *variable store*. Variables are then accessed as required by index, and to aid readability of the code, `VARX` is aliased to `int` 0, `VARY` to 1, `VARZ` to 2, and so on for `VARU, VARV, VARW, VARG, VARD, VARE` (defined in enumeration `AXIOMVAR`). This array is available to any function in the eml-module. It can be accessed via `eml_GetVar()`. When `eml_GetVar()` is used, if the variable is not in the variable store, it is created by calling `symbol_CreateStandardVariable()`.

Table 6.2. Functions Manipulating Propositional Variables

`SYMBOL` **`eml_GetVar`**`(int Index)`

6.1.3 Symbols:

The binary predicate symbol representing the accessibility relation (for example $R_r(x,y)$) or the unary predicate symbol uniquely associated with the a particular modal formula (for example Q_ϕ for modal formula ϕ), are stored in a cache which is local to the eml-module. This cache was developed in response to problems with dynamic allocation of symbols within SPASS, and to reduce overheads in retrieving symbols that already exist. The structure of the *symbol cache* is an array (`eml_AxiomSymbolList[]`) of type `AxiomSymbol. AxiomSymbol` is a `struct` with elements as follows.

- `char * Name` – A String (`char *`) based representation of the `TERM`
- `SYMBOL Symbol` – The actual symbol
- `TERM Term` – A copy of the `TERM` that generated the symbol name
- `eml_SYMBOLTYPE Type` – `eml_AXIOM` (for Q_p) or `eml_AXIOMREL` (for R_r)
- `LIST NegBox` – Used during setting precedence, see section 6.10.
- `int AccessCount` – Used during setting precedence, see section 6.10.

A maximum of `eml__MAXNOAXIOMSYMBOL` entries can be stored in the cache, and the next empty slot in the array is held in `eml_INDEXAXIOMSYMBOL`. These symbols are retrieved in response to a request for the symbol by the String-based representation of the modal term to `eml_CreateAxiomSymbol()`. If possible the symbol is retrieved from SPASS (using `symbol_Lookup()`) or from the local symbol cache (`eml_GetAxiomSymbol()`). When a new symbol must be created it is added to the symbol cache by `eml_SetAxiomSymbol()`. The symbol itself is created by `symbol_CreatePredicate()` in `eml_CreateAxiomSymbol()`. Only two `eml_SYMBOLTYPES` are needed; `eml_AXIOM` and `eml_AXIOMREL`, corresponding to `eml_PREDNAMEPREFIX[SymbolType]`s "Q" and "R", and with arity of 1 and 2 respectively. The default precedence of symbols is the default precedence allocated by SPASS (simply corresponding to the order in which they are created), but is unimportant since the precedence is usually modified at exit from the eml-module (see section 6.10). Symbol names are of maximum length `symbol__SYMBOLMAXLEN(64)`. Names longer than this are modified to a unique name reflecting the (arbitrary) position in the local cache (for example, perhaps Q_{28} renamed from `Qnot_and_and_box_r_p_not_box_r_box_r_box_r_not_p_not_box_r_box_r_box_r_box_r_box_r_box_r_not_q`). The user is informed of the correspondence between the two names. In order to make the `TERM`s created in the translation easily traceable to a specific origin, names that would be smaller than `symbol__SYMBOLMAXLEN` are still presented in the rich format. This can be over-ridden by setting the option `HiddenOpt.AlwaysUseSubstitutedSymbolNames` so that all symbols are named in the format Q_{int}.

Functions involving creation and retrieval of these symbols are assisted by several utility functions, as follows.

- `eml_TermName()` / `eml_SymbolNameHelper()` – A String (that is, `char` pointer) is modified by a recursive function to contain the names of all the symbols contained in the input `TERM`. Implementation of these functions required small changes to the symbol-module: addition of the function `symbol_GetSymbolSTDVARIABLENAMES()` and `symbol_PrecedenceToList()` which give access to String representation of variables.
- `eml_TermLength()` / `eml_TermLengthHelper()` - Recursively counts the number of symbols in a `TERM`.

Several optimizations (section ?) are implemented in which (i) `TERM`s have double negation eliminated `eml_EliminateNotNot()`, (ii) conjunctive `TERM`s are re-ordered in a standard way using a merge sort (`list_MergeSort()` using the comparator `eml_TermNameTest()`), which compares the sub-`TERM`'s String representation (`strcmp()` on the name) before naming the entire `TERM` (`eml_OptTermName()` using utilities `eml_TermNameSort()`, `eml_SortTermList()`). A trivial example would re-order $c \wedge b \wedge a$ to $a \wedge b \wedge c$ before generating the `TERM` name. This option can be disabled using `HiddenOpt.DoNotSortTermNames`.

This name allocated to a `TERM` is used in determining whether `TERM`s are equal in the function `eml_TermEqual()`. The function `eml_TermCompare()` compares `TERM`s in the same way, and the integer returned follows the semantics of `strcmp(name1, name2)`.

Table 6.3. Functions Manipulating Symbols

`SYMBOL` **`eml_GetAxiomSymbol`**`(char *Name)`
`void` **`eml_SetAxiomSymbol`**`(char *Name, SYMBOL *, TERM, eml_SYMBOLTYPE)`
`SYMBOL` **`eml_CreateAxiomSymbol`**`(TERM,eml_SYMBOLTYPE,int Arity,int Status, PRECEDENCE)`
`int` **`eml_TermLength`**`(TERM)`
`void` **`eml_TermLengthHelper`**`(TERM, int *Length)`

char ***eml_TermName**(TERM, char *str)
char ***eml_SymbolNameHelper**(SYMBOL, char* str)
char ***eml_OptTermName**(TERM, char *str)
TERM **eml_TermNameSort**(TERM)
LIST **eml_SortTermList**(LIST)
BOOL **eml_TermNameTest**(POINTER Ptr1, POINTER Ptr2)
int **eml_TermCompare**(TERM Term1, TERM Term2)
BOOL **eml_TermEqual**(TERM Term1, TERM Term2)

6.1.4 The Instantiation Set.

The current instantiation set is stored in the struct AXIOMDEFLIST. The instantiation list is passed in function augments when required.

- `TERM Term` - The original `TERM`.
- `LIST SuperDef` - List of pointers to *And* (and possibly *Not*) `TERM`s.
- `LIST BoxDef` - List pointers to *Box* `TERM`s.
- `LIST SubDef` - List of pointers to *predicate* `TERM`s.
- `LIST XtraBoxDef` - List of compositional `TERM`s incremented as axioms are processed.
- `LIST CompositionList` - List of compositional `TERM`s in temporary storage.

Only one instantiation set is formed during the translation, Defs, that arises in `eml_Axiomatic()`. This instantiation set is formed by recursive analysis of the standardized input `TERM` by the functions `eml_GetDEFUnits()` / `eml_GetDEFUnitsHelper()` – see also section 6.8. Three lists of `TERM`s (containing pointers to `TERM`s) are generated arising from sub`TERM`s beginning with a *predicate* (`AXIOMDEFLIST.SubDef`), with a *box* symbol (`AXIOMDEFLIST.BoxDef`), and with an *and* symbol (`AXIOMDEFLIST.SuperDef`). If the option IncludeNegativeDefs is enabled, then the list SuperDef also contains `TERM`s beginning with a *not* symbol. In order to avoid duplication, (and hence wasted time in the resolution procedure), `TERM`s that are already present in a particular list are not added a second time to the instantiation set. XtraBoxDef contains compositional `TERM`s accumulated during the translation of modal axioms, corresponding to the additional `TERM`s contributing to the *current* instantiation set. Compositional `TERM`s are stored temporarily in the CompositionList, before being processed and moved to the `XtraBoxDef`. This is the mechanism whereby, for example, the compositional `TERM`s arising from one modal axiom are only used when the *next* axiom is translated..

Several utilities are involved in administration of this data structure. `eml_listContains()` loops through the list of `TERM`s, and returns TRUE if the query `TERM` is already in the list. Comparison is based on the (sorted) string representation of the name of the `TERM` (in `eml_TermEqual()`). `eml_AxiomDEFListConc()` returns a concatenated list (in the mode is `eml_ALL` it returns all of `BoxDef` + `SubDef` + `SuperDef`; in the mode `eml_COMPLEX` only `BoxDef` + `SuperDef` are returned). `eml_AxiomDEFListInit()` initializes a blank `AXIOMDEFLIST` structure.

Table 6.4. Functions Manipulating the Instantiation set.

void **eml_GetDEFUnits**(AXIOMDEFLIST *)
void **eml_GetDEFUnitsHelper**(TERM Term, AXIOMDEFLIST *DEFs)
LIST **eml_AxiomDEFListConc**(AXIOMDEFLIST *DEFs, EML_CONTROL Mode)
BOOL **eml_ListContains**(LIST *List, TERM Term)
BOOL **eml_TermEqual**(const TERM Term1, const TERM Term2)

6.1.5: Other Data:

The array `eml_R[]` stores each modality index found in the input problem, the with the next free slot in the array (and number of modalities found) stored in `eml_INDEX_R` . It is populated just before the Axioms defined are processed in `eml_AxiomLogicControl()` by calling `eml_Count_Modalities()`.

In order to avoid duplication in the '`FINAL TERM`' submitted to the SPASS resolution prover, a `LIST` is maintained in `AlreadyDEFinedList`, of `TERM`s which have been previously DEFined in `eml_DEF()`/`eml_SimpleDEF()` against which new `TERM`s can be screened (eml_ListContains()). The LIST is added to at the end of `eml_SimpleDEF()` by the function `eml_ListAddTerm()`.

6.2 : Utility Functions:

Throughout the translation, comprehensive output is printed to `standard output` for the information of the user. Utility functions are used to print information under control of the parameter

`HiddenOpt.DoNotUsePrintStatements`, which can be activated (`eml_ON`) to bypass the printing of output. A new function is provided to print a String in combination with each of one or more other parameters. The printing mode chosen for `TERM`s uses prefix notation (`fol_PrintDFG()`), rather than the internal format of `SPASS` (`term_Print()`). The printing functions are listed in table 6.5. A similar pattern is seen for the '`Error`' functions, which when called, flush a String of information to `standard output`, and then immediately terminate execution.

Other utility functions are used throughout the code. They include a function that adds a `TERM` to a `LIST` and return the now larger `LIST` (`eml_ListAddTerm()`), a similar function for adding `LIST`s together (`eml_ListAddList()`), and a function reporting whether a `LIST` contains a particular `TERM` (`eml_ListContains()`). The comparison of `TERM`s (in `eml_TermEqual()`) is based upon comparison of the String names associated with the `TERM`s by functions `eml_OptTermName()` or `eml_TermName()`, see section 6.1.3). `eml_QuantifyTerm()` helps the developer create the syntax to enclose a `TERM` in either a `fol_All()` or a `for_Exist()` quantifier.

Table 6.5. Miscellaneous Utility Functions:

void **eml_Print**(char *Str)
void **eml_PrintD**(char *Str,int)
void **eml_PrintS**(char *Str,SYMBOL)
void **eml_PrintDS**(char *Str,int,SYMBOL)
void **eml_PrintT**(char *Str,TERM)
void **eml_PrintDT**(char *Str,int,TERM)
void **eml_Error**(char *Str)
void **eml_ErrorD**(char *Str,int)
void **eml_ErrorC**(char *Str1, char *Str2)
LIST **eml_ListAddTerm**(TERM AddedTerm, LIST AddeeList)
LIST **eml_ListAddList**(LIST AddedList, LIST AddedList)
BOOL **eml_ListContains**(LIST *,TERM)
BOOL **eml_TermEqual**(TERM term1, TERM term2)
TERM **eml_QuantifyTerm**(SYMBOL Quantifier,TERM,LIST VarListContents)

In the following sections, the implementation of the complete set of tasks taken from figure ? is described.

6.3. Initialize EML module:

Some initialization of the eml-module was already in-place at the start of this project. For example, symbols like `box` and `dia` are already defined (`eml_Init()`). `eml_Init()` is called before any other function in the eml-module, from `top.c`. For this study new in-line modal-axiom symbols (section ?), and other new extensions to the dfg syntax like `slf()` and `noAxiom()`, are defined in the same way. These `SYMBOL`s need to be defined before the axiomatic translation starts in order that they can be related to the statements found in the .dfg file during parsing. Each symbol is introduced in the `dfgscanner.l` file, where it is associated (indirectly) with the name of the `SYMBOL` in `eml.c` and `eml.h` (defined both locally in `eml.c` and as an `extern SYMBOL` in `eml.h`). For example, `axD` (corresponding to the in-line dfg syntax element `axD()`) is defined to return `DFG_AXIOMD` in `dfgscanner.l`. In turn, `DFG_AXIOMD` is then associated with the function `eml_Axiom_D(void)` (from `eml.h`) in the file `dfgparser.y`. The precise syntax applicable to each symbol is also defined in `dfgparser.y`. The function `eml_Axiom_D(void)` returns a `SYMBOL eml_AXIOM_D` (defined both in `eml.c` and `eml.h` as described above). This `SYMBOL` is defined as a binary junctor using `symbol_CreateJunctor()` in `eml_Init()`, where the name `axD` is again associated with the `SYMBOL eml_AXIOM_D`. Many simple functions allowing these `SYMBOL`s to be recognized (for example, `eml_IsAxiom_D_Symbol(SYMBOL)` which returns `TRUE` for the `SYMBOL eml_AXIOM_D`) and grouped together (for example, `eml_IsAxiomTypeSymbol(SYMBOL) which returns TRUE` for any of the `SYMBOL`s of the same type as `eml_AXIOM_D`) are defined in `eml.h`. A full list of symbols is given in table 6.6.

Table 6.6: Correspondence between dfg syntax and internal symbols in eml.c/eml.h

SYMBOL defined in `eml.c/eml.h`	dfg syntax defined in `dfgscanner.l/dfgparser.y`	SYMBOL defined in `eml.c/eml.h`	dfg syntax defined in `dfgscanner.l/dfgparser.y`

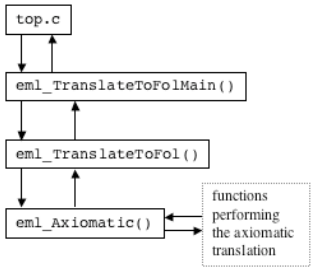
eml_AXIOM_D	axD
eml_AXIOM_B	axB
eml_AXIOM_T	axT
eml_AXIOM_4	ax4
eml_AXIOM_5	ax5
eml_AXIOM_ALT1	axALT1
eml_AXIOM_Dc	axDc
eml_AXIOM_Bc	axBc
eml_AXIOM_Tc	axTc
eml_AXIOM_4c	ax4c
eml_AXIOM_5c	ax5c
eml_AXIOM_ALT1c	axALT1c
eml_AXIOM_Do	axDo
eml_AXIOM_Bo	axBo
eml_AXIOM_To	axTo
eml_AXIOM_4o	ax4o
eml_AXIOM_5o	ax5o
eml_AXIOM_ALT1o	axALT1o
eml_AXIOM_4K2	ax4K2
eml_AXIOM_4K3	ax4K3
eml_AXIOM_5K2	ax5K2
eml_AXIOM_5K3	ax5K3

eml_AXIOM_ALT1KK11	axALT1KK11
eml_AXIOM_ALT1KK12	axALT1KK12
eml_AXIOM_ALT1KK21	axALT1KK21
eml_AXIOM_ALT1KK22	axALT1KK22
eml_AXIOM_4K2c	ax4K2c
eml_AXIOM_4K3c	ax4K3c
eml_AXIOM_5K2c	ax5K2c
eml_AXIOM_5K3c	ax5K3c
eml_AXIOM_ALT1KK11c	axALT1KK11c
eml_AXIOM_ALT1KK12c	axALT1KK12c
eml_AXIOM_ALT1KK21c	axALT1KK21c
eml_AXIOM_ALT1KK22c	axALT1KK22c
eml_AXIOM_4K2o	ax4K2o
eml_AXIOM_4K3o	ax4K3o
eml_AXIOM_5K2o	ax5K2o
eml_AXIOM_5K3o	ax5K3o
eml_AXIOM_ALT1KK11o	axALT1KK11o
eml_AXIOM_ALT1KK12o	axALT1KK12o
eml_AXIOM_ALT1KK21o	axALT1KK21o
eml_AXIOM_ALT1KK22o	axALT1KK22o
eml_SLF	slf
eml_NOAXIOM	noAxiom

The remainder of the initialization for axiomatic translation does not need to takes place until immediately before beginning the translation process, in the function `eml_AxiomInit()`. For example, the variable store `eml_AxiomVarStore[]` is initialized, and K is made the first modal-axiom in the axiom cache, and the two bit-based options flags are processed using `eml_AxiomOptionsInit()`. It makes more sense to discuss this latter function in the context of analysis of the other flags. Initialization via `eml_AxiomInit()` is called as the first action in the function `eml_Axiomatic()`.

Table 6.7. Initialization

void eml_Init(PRECEDENCE)
void eml_AxiomInit(FLAGSTORE, PRECEDENCE)
void eml_AxiomOptionsInit(FLAGSTORE)



6.4. Enter EML module from SPASS:

The eml-module is entered from `top.c` via `eml_TranslateToFolMain()`. This route was already defined in SPASS. The eml-module is directed to access the axiomatic translation routines from `eml_TranslateToFolMain()` by detection of activated `flag_EMLAXIOM==1`. The sequence of functions called is `eml_TranslateToFOL()`, and then `eml_Axiomatic()`.

6.5. Reorganize Axioms and Conjectures of input problem:

The `LIST*s AxiomList` and `ConjectureList` are passed to the eml-module from `top.c`. The preferred mode of delivery of the input problem is in a single member of the `AxiomList`, with no entries in the `ConjectureList`. Other modes of input are allowed, with an arbitrary number of formulae in the axioms and conjectures sections. The `TERM` in each member of each `LIST` is rolled-up into a single composite `TERM` of the form $(A_1 \wedge A_2 \wedge \ldots A_n) \wedge (\neg C_1 \wedge \neg C_2 \wedge \ldots \neg C_n)$. Note, that the conjecture `TERM`s have already been automatically negated by this stage. This composite `TERM` is submitted to translation. The resulting `TERM` is used to replace the `TERM` in the first element of the `AxiomList`, with all the remaining elements in both lists being discarded (inside `eml_TranslateToFolMain()`). In the case where no axioms are supplied in the input problem, a dummy `AxiomList` is formed prior to this manipulation by copying the contents of the `ConjectureList`. The output of the translation process always produces a standard format; that is a single member of the `AxiomList`. At each stage the user is informed of the transformations taking place.

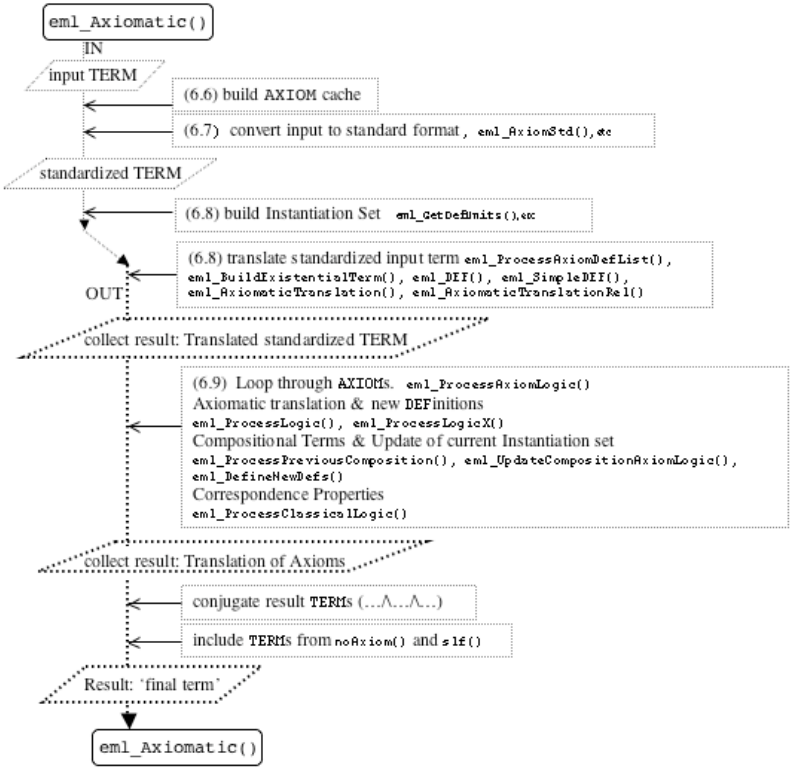
The composition of the input problem is also used to set the flag `DoNotUseLocalSatisfiability`. If only axioms are present then `DoNotUseLocalSatisfiability` is `eml_ON`, defining use of the global satisfiablity mode. If conjectures are present (with or without axioms) then the local satisfiablity mode is defined (`DoNotUseLocalSatisfiability` is `eml_OFF`). The choices are made based upon the `list_Length()` of the `ConjectureList` and `AxiomList`. The user is informed of the choices made.

The translation itself is called through `eml_TranslateToFol()`, with a positive polarity (since a single `AXIOM` is submitted), and subsequently via the function `eml_Axiomatic()`.

Table 6.8. Implementation of Tasks: Reorganize Axioms and Conjectures of input problem

void **eml_TranslateToFolMain**(LIST* AxiomList,LIST* ConjectureList, BOOL KeepProofSearch, FLAGSTORE, PRECEDENCE)
TERM **eml_TranslateToFol**(TERM,int Polarity,FLAGSTORE,PRECEDENCE)
TERM **eml_Axiomatic**(TERM Term,FLAGSTORE,PRECEDENCE)

The hub of control of the axiomatic translation is in the function `eml_Axiomatic()`. All subsequent processes in the axiomatic translation are called directly or indirectly from here, and eventually return their results back to this function.



6.6. Read and analyze .dfg file:

This syntax was implemented by extensive additions to the source files *dfgscanner.l* and *dfgparser.y*, which serve as input for the Bison and Flex code generating tools. Details are given below.

Bit-based options flags:

Control over the translation process can be exercised by using the bit-based options flags. Two new flags have been added to extend the existing SPASS flags mechanism (named `EMLAxiomOpt` and `EMLAxiomHidOpt` in the dfg syntax, and `flag_EMLAXIOMOPTIONS` and `flag_EMLAXIOMHIDDENOPTIONS` internally). Bit-wise flags are a new concept in SPASS. They allow the user to supply eight options per flag. SPASS flags are integers. In these cases a nine-digit integer is entered, beginning with a 'dummy' digit 9, with the other digits generally being 0 (off) and 1 (on). Occasionally other digits are used to represent more complex options. The bit-based flags are

processed in the function eml_AxiomOptionsInit() that is called during the initialization of the axiomatic translation from eml_AxiomInit(). Each option is represented by a member of the structure HiddenOpt(an instance of the struct AXIOMHIDDENOPTION called HiddenOpt), each named according to the functionality that it controls, and taking a value from the enumeration EML_CONTROL (with allowed values eml_ON, eml_OFF, eml_FULL, eml_ALL, eml_COMPLEX, eml_LOCAL, eml_GLOBAL). These members of HiddenOpt are used in such a way that the default state (which does not change the default functioning of the axiomatic translation) is always eml_OFF. This initial state is set in eml_AxiomOptionsInit(), and then may be overridden if the corresponding bit is set in the bit-wise flags. The values assigned to the bit-wise flags themselves are analyzed by the function eml_Int2String(). In effect, the values stored in the bits are transferred into a more easily accessed cache (HiddenOpt). Whenever a bit is set to a value other than the default, the user is informed of the option that has been chosen. This options cache is also used to store some control parameters resulting from analysis of the problem by the software (that is, not under user control; for example, HiddenOpt.UsingInlineAxiomfeatures). The bits available and the corresponding options that are controlled by them are listed in table 6.10.

Table 6.9. Implementation of Tasks: Control Flags. Read and analyze .dfg file

void eml_Int2String(int Value, char *Integer)
void eml_AxiomOptionsInit(FLAGSTORE)

Table 6.10. Details of the correspondence between bitwise flags and HiddenOpt parameters

User interface flag	Processed version of flag – local to eml-module
EMLAxiomOpt[0]==1	HiddenOpt.DoNotUseAxiomBComposition==eml_ON
EMLAxiomOpt[1]==1	HiddenOpt.DoNotUseAxiomDComposition==eml_ON
EMLAxiomOpt[2]==1	HiddenOpt.DoNotUseAxiomAlt1Composition==eml_ON
EMLAxiomOpt[3]==1	HiddenOpt.DoNotUseAxiom4Composition==eml_ON
EMLAxiomOpt[4]==1	HiddenOpt.OverrideLocalGlobal==*eml_GLOBAL*
EMLAxiomOpt[4]==2	HiddenOpt.OverrideLocalGlobal==*eml_LOCAL*
EMLAxiomOpt[5]==1	HiddenOpt.DoNotUseAxiom4Kcomposition==eml_ON
EMLAxiomOpt[6]==1	HiddenOpt.DoNotUseAxiom5Kcomposition==eml_ON
EMLAxiomOpt[7]==1	HiddenOpt.DoNotUseAxiomAlt1KKComposition==eml_ON
EMLAxiomHidOpt[0]==1	HiddenOpt.DoNotUseSetPrecedence==eml_ON
EMLAxiomHidOpt[1]==1	HiddenOpt.IgnoreLogicFlagWarnings==eml_ON
EMLAxiomHidOpt[2]==1	HiddenOpt.ExcludeOptionalPositiveShortcuts==eml_ON
EMLAxiomHidOpt[3]==1	HiddenOpt.DoNotSortTermNames==eml_ON
EMLAxiomHidOpt[4]==1	HiddenOpt.DoNotUseComposition==eml_ON
EMLAxiomHidOpt[5]==1	HiddenOpt.AlwaysUseSubstitutedSymbolNames==eml_ON
EMLAxiomHidOpt[6]==1	HiddenOpt.DoNotUsePrintStatements==eml_ON
EMLAxiomHidOpt[7]==1	HiddenOpt.NeverUsePositiveShortcuts==eml_ON
Set by the software (not by the user)	HiddenOpt.IncludeNegativeDefs
	HiddenOpt.UsingInlineAxiomfeatures
	HiddenOpt.UsingSetFlagAxiomfeatures
	HiddenOpt.DoNotUseLocalSatisfiability

Modal Axioms:

The eml-module holds the requested modal axioms for the axiomatic translation in the axiom cache. This has already been discussed (section 6.1.1). There are many mechanisms by which the user may input modal axioms, and the control statements for processing of these dfg syntaxes is contained early in the function eml_Axiomatic(). Each syntax populates the *same* axiom cache. The axiom definition mechanisms are processed in the order (i) in-line syntax (ii) set_axiom() syntax and (iii) set_flag() or command-line switch syntax. If axioms are defined at any stage (eml_AreAxiomsDefined()), then subsequent processing is skipped, giving a precedence to the different forms of syntax. Various internal flags are set eml_ON according to the syntax that has been chosen, for example HiddenOpt.UsingInlineAxiomfeatures or HiddenOpt.UsingSetFlagAxiomFeatures. Finally, contents of the axiom cache is printed for the information of the user by the function eml_PrintAxiomComposition(). The details of the processing of the individual syntaxes for requesting modal axioms are given below.

Table 6.11: Implementation of modal axiom syntaxes.

void **eml_PrintAxiomComposition**()
BOOL **eml_AxiomsAreClassical**()

BOOL **eml_AreAxiomsDefined**()

In-line syntax for modal axioms:

The definition of symbols for the in-line syntax has already been described (section 6.3, eml_Init()). The analysis in-line syntax is implemented in eml_ComposeAxioms() (called from eml_Axiomatic()) which recursively scans through the input TERM, stripping away the in-line syntax components, and building the AXIOM cache (using eml_BuildAxiomR()).

Table 6.12. Implementation of Inline Syntax

TERM **eml_ComposeAxioms**(TERM)
BOOL **eml_IsAxiomTypeSymbol**(SYMBOL S)
BOOL **eml_IsAxiom_X_Symbol**(SYMBOL S)

Extensions to the standard flag mechanism in SPASS:

The simplest new input mechanism to implement was the new standard flags to added to SPASS. All that is required are formulaic modifications to flags.c and flags.h, following the same pattern already used for other flags of the type flag_EMLSPECIAL. In eml.h, the flags all need to be defined in the enumeration FLAG_ID (these are listed in table 6.13), and properties of each individual flag (for example, the maximum and minimum numerical values allowed) are enumerated (for example, in the enum FLAG_EMLAXIOMLOGICDTYPE for flag_EMLAXIOMLOGICD). In eml.c, for each individual flag_EMLSPECIAL flag, the function flag_InitIntern() is used to associate those values just defined (above) with the flag-name used in the command-line switch of the dfg syntax, for example, EMLAxiomLogD. A full list of these flags is given in table 6.13.

These flags are analyzed in the function eml_AnalyzeEmlAxiomTranslationFlags() building the AXIOM cache using functions eml_BuildAxiomAt() and eml_BuildAxiom(). The first axiom defined is always K. In the standard mode, checking is performed to restrict allowed other modal-axioms to those seen in reference [1] (None, 4, 5, T, B, D, alt_1, 4^k, 5^κ, $alt_1{}^{\kappa 1\kappa 2}$, T.4, T.B, D.B, D.4, 4_O.B, 5_O.B, T.5_O, T.4_O.B, D.4.B; T_C, D_C, 5_C, 4_C, B_C, $alt_1{}_C$, $4_C{}^k$, $5_C{}^\kappa$, $alt_1{}_C{}^{\kappa 1\kappa 2}$ and D_C.B, D_C.4, T_C.4.B_C; axioms 4^κ, 5^κ, $4_C{}^\kappa$ & $5_C{}^\kappa$ currently allow κ = 2 or 3; axiom $alt_1{}^{\kappa 1\kappa 2}$ & $alt_1{}_C{}^{\kappa 1\kappa 2}$ options currently allow $\kappa 1$ or $\kappa 2$ = 1 or 2). The value of any particular flag is accessed with the function flag_GetFlagValue(). Axioms are marked as active by giving the appropriate flag a value greater than 1. In the *experimental* mode, the option IgnoreLogicFlagWarnings has been enabled (eml_ON), and *any* combination of modal-axioms is allowed. In this mode, the value assigned by the user, to the flag is used to sort the added modal-axioms. Modal-axioms assigned lower values are added to the axiom cache first. If flags have the same value, then the ordering is determined by the arbitrary order in which testing for a particular flag occurs in the C-code. The function eml_AnalyzeEmlAxiomTranslationFlags() also implements the control over the translation by the option flags that exclude composition from the translation (Hidden.DoNotUseComposition applied to all axioms; or for individual axioms Hidden.DoNotUseAxiomBComposition, etc…).

Table 6.13. New flags implemented in extended SPASS to support Axiomatic Translation

Internal name of flag	dfg command-line name of Flag	Maximum value	Minimum value
flag_EMLAXIOM	EMLAxiom		-1
flag_EMLAXIOMOPTIONS	EMLAxiomOptions	999999999	-1
flag_EMLAXIOMHIDDENOPTIONS	EMLAxiomHidOpt	999999999	-1
flag_EMLAXIOMLOGIC4	EMLAxiomLog4	9	-1
flag_EMLAXIOMLOGIC5	EMLAxiomLog5	9	-1
flag_EMLAXIOMLOGICB	EMLAxiomLogB	9	-1
flag_EMLAXIOMLOGICT	EMLAxiomLogT	9	-1
flag_EMLAXIOMLOGICD	EMLAxiomLogD	9	-1
flag_EMLAXIOMLOGICALT1	EMLAxiomLogAlt1	9	-1
flag_EMLAXIOMLOGIC4K	EMLAxiomLog4K	9	-1
flag_EMLAXIOMLOGICFACTOR4	EMLAxiomLogFac4	9	-1
flag_EMLAXIOMLOGIC5K	EMLAxiomLog5K	9	-1
flag_EMLAXIOMLOGICFACTOR5	EMLAxiomLogFac5	9	-1
flag_EMLAXIOMLOGICALT1KK	EMLAxiomLogAlt1KK	9	-1
flag_EMLAXIOMLOGICFACTOR1	EMLAxiomLogFac1	9	-1
flag_EMLAXIOMLOGICFACTOR2	EMLAxiomLogFac2	9	-1
flag_EMLCLASSLOGIC4	EMLClassLog4	9	-1
flag_EMLCLASSLOGIC5	EMLClassLog5	9	-1
flag_EMLCLASSLOGICB	EMLClassLogB	9	-1
flag_EMLCLASSLOGICT	EMLClassLogT	9	-1

flag_EMLCLASSLOGICD	EMLClassLogD	9	-1
flag_EMLCLASSLOGICALT1	EMLClassLogAlt1	9	-1
flag_EMLCLASSLOGIC4K flag_EMLCLASSLOGICFACTOR4	EMLClassLog4K EMLClassLogFac4	9 9	-1 -1
flag_EMLCLASSLOGIC5K flag_EMLCLASSLOGICFACTOR5	EMLClassLog5K EMLClassLogFac5	9 9	-1 -1
flag_EMLCLASSLOGICALT1KK flag_EMLCLASSLOGICFACTOR1 flag_EMLCLASSLOGICFACTOR2	EMLClassLogAlt1KK EMLClassLogFac1 EMLClassLogFac2	9 9 9	-1 -1 -1

Table 6.14. Implementation SPASS-type Axiom flags. Read and analyze .dfg file

void **eml_AnalyzeEmlAxiomTranslationFlags**(FLAGSTORE)
void **eml_PrintAxiomComposition**()

The `set_axiom(`*`modalityIndex, Axiom`*`)` syntax.

The modifications required to support the `set_axiom()` syntax are quite extensive. In `dfgscanner.l`, the dfg syntax `set_axiom()` is defined to return `DFG_SETAXIOMFLAG`. In `dfgparser.y`, the String content of `set_axiom()` is extracted (borrowing the syntax from the pre-existing `set_flag/DFG_SETFLAG` mechanism). The content is parsed into a list (`axiompairlist`) of elements of type `axiompair`. An `axiompair` element can adopt three formats. In most cases (other than requests for bimodal axioms, and for axioms `4` and `5`), the each element is a pair `[id,DFG_ID]`, with a symbol defined elsewhere in the dfg file representing the modality index to which this modal-axiom will be applied (`id`) and `DFG_ID` being a String representing the modal-axiom being requested (that may have the format of a series of characters `[a-zA-Z0-9_]+`). In this case, the content is processed by the function `eml_SetAxiomFlagValue(id, DFG_ID)`. For axioms 4 and 5, the format of the pair is `[id,number]` (with `number` restricted to the format `[0-9]+` for axioms 4 and 5), and the content is processed by the function `eml_SetAxiomFlagValue2(id,number)`. For bimodal axioms, the format of the "pair" is `[id`$_1$`,id`$_2$`,DFG_ID]`, with the two modal indices being defined, and the content processed by the function `eml_SetAxiomFlagValueBi(id`$_1$`,id`$_2$`,DFG_ID)`.

During the parsing process, the input is subject to error checking against symbols defined in the dfg file (see above), and against the allowed values of the axioms that can be requested (see next) by functions `eml_SetAxiomFlagValue()`, `eml_SetAxiomFlagValue2()` and `eml_SetAxiomFlagValueBi()`. As already mentioned, these functions respectively handle (i) all the modal-axioms *except* bimodal axioms and axioms 4 and 5, (ii) axioms 4 and 5 (different because an integer must be parsed rather than a String), and (iii) the bimodal axioms. The allowed values for axioms are checked against the possible values in `eml_AllowedSetFlags[]`. These are :

```
None, T, D, B, 4, 5, ALT1, To, Do, Bo, 4o, 5o, ALT1o, Tc, Dc, Bc, 4c, 5c, ALT1c, 4K2,
4K3, 5K2, 5K3, ALT1KK11, ALT1KK12, ALT1KK21, ALT1KK22,4K2o, 4K3o, 5K2o, 5K3o,
ALT1KK11o, ALT1KK12o, ALT1KK21o, ALT1KK22o, 4K2c, 4K3c, 5K2c, 5K3c, ALT1KK11c,
ALT1KK12c, ALT1KK21c, ALT1KK22c, SRo, Go, DENo, TRo, Mo, Wo, SR, G, DEN, TR, M, W,
DBBB, DBBBo, CRo, CR2o, B2o, B3o, CR, CR2, B2, B3, H, Ho, CR3, CR3o
```

These functions `eml_SetAxiomFlagValue()`, `eml_SetAxiomFlagValue2()` and `eml_SetAxiomFlagValueBi()` are also used to record the values in the `set_axiom()` parameter, by building `eml_AXIOMFLAG[]` array, with elements of type `AXIOMFLAG`. The elements of the `struct AXIOMFLAG` are:

- `char *R` - the modality index.
- `char *S` - the second modality index for bi-modal axioms.
- `char *Axiom` - the requested axiom name.

Within the `eml_AXIOMFLAG[]` array `eml_INDEX_FLAG` points to next available slot. Again, it is pointed out that this array is populated during parsing of the dfg file, before axiomatic translation is initiated.

After axiomatic translation begins, this array `eml_AXIOMFLAG[] is` analyzed by `eml_AnalyzeSetAxiom()` (called from `eml_Axiomatic()`) which creates the requested axiom cache using the functions `eml_BuildAxiomR()` and `eml_BuildAxiomRS()`. The order in which axioms were defined in the `set_axiom()` syntax is preserved in the `eml_AXIOMFLAG[] array`, and is used to assign the correct order in which to apply modal axioms in the translation of axioms.

Note that the arrangement described above isolates modifications as far as possible to the eml-module. In SPASS it would have been more usual to implement parameter checking (and other similar functionality) during dfg parsing *within* the `dfgparser.y` file, but extensive changes there would be more difficult to effectively isolate from pre-existing code.

Table 6.15. Implementation of set_axiom() mechanism.

void **eml_AnalyzeSetAxiom**(FLAGSTORE)
void **eml_SetAxiomFlagValue**(char *Value1, char *Value2)
void **eml_SetAxiomFlagValue2**(char *Value1, char *Value2)
void **eml_SetAxiomFlagValueBi**(char *Value1, char *Value3, char *Value2)

6.7. Standardization of input:

The functions listed in table 6.16 implement re-writing of the input `TERM` in standard format (see table 5.?). The functions are implemented to traverse the input `TERM` recursively. When modifications are made, the result is re-submitted to the function. The recursion is terminated when a non-complex (predicate) `TERM` is encountered. The function controlling this process is `eml_AxiomaticStd()`. The standardization functions are named according to the transformation that they carry out. A `BOOL`ean informs the calling function whether a modification has been made. In order to promote reuse of these functions, and to aid simplicity (and hence reliability) of the code, modifications are made successively, one-at-a-time. The alternative would be to make all the modifications in a single pass through the input `TERM`. Little additional overhead is incurred as compared to a single recursive traversal of the `TERM`. The order in which transformations are applied has already been described (section 5.3). A small number of transformations that were already present in the eml-module are applied at the beginning of the `eml_AxiomaticStd()`. (These re-used functions are marked with an asterix * in table 6.16).

Table 6.16. Functions associated with standardization of the input problem.

TERM **eml_AxiomaticStd**(TERM)	
TERM eml_RemoveTrivialAtomsAndOps(TERM) * *TERM eml_ObviousSimplifications(TERM,Flags)* *	
TERM **eml_EliminateModals**(TERM, *BOOL)	□⊤ ≡ ⊤ □¬⊥ ≡ ¬⊥ ◇⊥ ≡ ⊥ ¬□¬⊥ ≡ ⊥
TERM **eml_EliminateDiamond**(TERM, *BOOL)	◇p ≡ ¬□¬p
TERM **eml_EliminateTrue**(TERM, *BOOL)	⊤ ≡ ¬⊥
TERM **eml_EliminateBiImplication**(TERM, *BOOL)	p↔q ≡ (p→q)∧(p←q)
TERM **eml_EliminateImplied**(TERM, *BOOL)	p←q ≡ (q→p)
TERM **eml_EliminateImplies**(TERM, *BOOL)	p→q ≡ (¬p∨q)
TERM **eml_EliminateOr**(TERM, *BOOL)	(p∨q) ≡ ¬(¬p∧¬q) (p∨q∨r) ≡ ¬(¬p∧¬q∧¬r)……
TERM **eml_EliminateNotNot**(TERM, *BOOL)	¬¬p ≡ p
TERM **eml_NegateElimNotNot**(TERM) TERM **eml_Negate**(TERM)	p becomes ¬p; ¬p becomes p p becomes ¬p
TERM **eml_EliminateNestedAnd**(TERM, *BOOL) TERM **eml_EliminateBoxAnd**(TERM, *BOOL) TERM **eml_EliminateSingletAnd**(TERM, *BOOL)	∧(p,∧(a,b),q,p) ≡ ∧(p,a,b,q) □(p∧q) ≡ (□p∧□q) ∧(p) ≡ p

6.8 Perform Axiomatic Translation of input problem:

The axiomatic translation requires an *instantiation set* of `TERM`s. The instantiation set from the original input `TERM` is found in an instance of `AXIOMDEFLIST` called `DEFs`, originating in `eml_Axiomatic()`. It is formed by recursive analysis of the standardized input `TERM` within the functions `eml_GetDEFUnits()` / `eml_GetDEFUnitsHelper()`. `DEFs` contains three lists of `TERM`s (`BoxDef`, `SubDef`, `SuperDef`). These three lists contain pointers to `TERM`s arising from sub`TERM`s beginning with a *predicate* (`DEFs.SubDef`), with a *box* symbol (`DEFs.BoxDef`), and with an *and* symbol (`DEFs.SuperDef`). If the option `HiddenOpt.IncludeNegativeDefs` is enabled, then the list `DEFs.SuperDef` also contains `TERM`s beginning with a *not* symbol. During construction of `DEFs`, a new `TERM` is not added to the instantiation set if it was already in the relevant list (tested with `eml_ListContains()`).

As the axiomatic translation progresses, the contributions of various procedures to the final translated `TERM` are accumulated in the `LIST OutList` within the controlling function `eml_Axiomatic()`. The translation of the default instantiation set is performed by the function `eml_ProcessAxiomaticDefList()` called from `eml_Axiomatic()`. In `eml_ProcessAxiomaticDefList()`, first the formula $\exists x Q \phi(x)$ from 3.xx is formed in by

calling the function eml_BuildExistentialTerm(), extracting DEFs.Term as the source of the name of the predicate Qϕ. This TERM may be universally quantified if global satisfiablity mode is selected (HiddenOpt.DoNotUseLocalSatisfiability is eml_ON). Special handing is needed for translation of the TERM false.

Following this, in eml_ProcessAxiomaticDefList(), each member of the instantiation set is translated according to formulae 3.x. For efficiency reasons, the translation is split into two. (The list of instantiation set TERMs is subdivided using the function eml_AxiomDEFListConc()). The members of the instantiation set other than simple predicates are translated by the function eml_DEF(). Members of the instantiation set that are simple predicates are translated by the function eml_SimpleDEF(). At these points, a decision is made about whether positive shortcuts are to be included in the translation, by calling the function eml_SePositiveShortcut() / eml_SetPositiveShortcutHelper(). In the default case, positive shortcuts are included. The results of the translation of each member of the instantiation set is accumulated in a LIST that is returned from eml_ProcessAxiomaticDefList() to eml_Axiomatic().

The function eml_DEF() creates a TERM of the form (for each individual member of the instantiation set) $\forall x(Q\phi(x)\rightarrow\pi(\phi,x)) \wedge \forall x(Q\phi(x)\leftrightarrow\neg Q\neg\phi(x)) \wedge \forall x(Q\neg\phi(x)\rightarrow\pi(\neg\phi,x))$. The parameters x and y (see formulae 3.xx) are passed to eml_DEF() from the variable store as VARX and VARY. In eml_DEF(), the four (three if positive shortcuts are excluded) subcomponents of the translation (see formulae 3.xx) are built up separately. Two of these subcomponents require translation of a fragment of the member of the instantiation set under consideration (the formal parameter Def for eml_DEF()) to be processed with function π as described in 3.xx – see the next paragraph. Note eml_DEF() and eml_SimpleDEF() are called in other contexts (during processing of modal axioms) when new symbols need to be DEFined.

The function π (as defined in formulae 3.3 and 3.3.1) is implemented in the function eml_AxiomaticTranslation(). A Def TERM is transformed according to the topmost symbol or operator, as listed below.

a. $\pi(\text{false}, x)$	returns false
b. $\pi(p, x)$	returns true
c. $\pi(\neg\phi, x)$	returns $\neg Q\phi(x)$
d. $\pi(\phi\wedge\psi, x)$	returns $Q\phi(x)\wedge Q\psi(x)$. Note: works correctly for the non-binary case returning $Qp(x) \wedge Qq(x) \wedge$ etc…
e. $\pi(\neg(\phi\wedge\psi), x)$	returns $Q\neg\phi(x)\vee Q\neg\psi(x)$. Note: works correctly for the non binary case reurning $Q\neg p(x) \vee Q\neg q(x) \vee$ etc…
f. $\pi(\Box(r,\phi), x)$	returns $\forall y(R(x, y) \rightarrow Q\phi(y))$
g. $\pi(\neg\Box(r,\phi), x)$	returns $\exists y(R(x, y) \wedge Q\neg\phi(y))$

In cases f and g, special handling is required for cases in which the TERM ϕ is false ($\Box$(r,false) and $\neg\Box$(r,false)). Variables (for example, x as VARX) are taken from the variable store. When a predicate of the form Qϕ is needed, it is created (or retrieved from the store) using the function eml_CreateAxiomSymbol(). In the cases f and g, the binary predicate corresponding to the accessibility relation (for example, $R_r(x,y)$) is obtained from the function eml_AxiomaticTranslationRel(). If translpairs are defined in the dfg file, then the assignment of the pair modality index/accessibility relation defined by the user is honored here (utilizing the pre-existing function eml_FoQuantAssocWithPropSymbol()).

Table 6.17. Implementation of Axiomatic Translation of input problem

void **eml_AxiomDEFListInit**()
void **eml_GetDEFUnits**(AXIOMDEFLIST *)
void **eml_GetDEFUnitsHelper**(TERM, AXIOMDEFLIST *)
TERM **eml_BuildExistentialTerm**(AXIOMDEFLIST *,SYMBOL VarX, FLAGSTORE,PRECEDENCE)
LIST **eml_ProcessAxiomaticDefList**(AXIOMDEFLIST *,FLAGSTORE,PRECEDENCE)
LIST **eml_DEF**(TERM Def,SYMBOL VarX,SYMBOL VarY,BOOL WithShortCut,FLAGSTORE, PRECEDENCE)
LIST **eml_SimpleDEF**(TERM Def, SYMBOL VarX, SYMBOL VarY, BOOL WithShortCut,FLAGSTORE,PRECEDENCE)
TERM **eml_AxiomaticTranslation**(TERM Def, SYMBOL VarX, SYMBOL VarY,FLAGSTORE,PRECEDENCE)
TERM **eml_AxiomaticTranslationRel**(TERM DefFragment,SYMBOL VarX,SYMBOL VarY, FLAGSTORE, PRECEDENCE)
BOOL **eml_SetPositiveShortcut**()
BOOL **eml_SetPositiveShortcutHelper**(Axiom)
LIST **eml_AxiomDEFListConc**(AXIOMDEFLIST *, EML_CONTROL Mode)

6.9. Perform Translation of the input problem for each modal Axiom in strict order:

If the function eml_AreAxiomsDefined() returns true and there are members of the instantiation set beginning with a *box* symbol, then the function eml_AxiomLogicControl() is called from eml_Axiomatic() to process the modal axioms that have been defined. eml_AxiomLogicControl() first updates the data in the eml_R[] array (see section 6.1.5), and if only one modality is present in the input TERM, and axioms have been defined via the command-line switches or the set_flag() mechanism, the Modality field is set for each member of AxiomComposition[] to the single modality found in the input problem. Then, looping through each AXIOM in the AxiomComposition[], the function eml_ProcessAxiomLogic() is called to process each AXIOM, and gather the output in a LIST for return.

Axiomatic Translation of modal axioms

In eml_ProcessAxiomLogic() the logic of the program flow has a modular design, but nevertheless is quite complicated. First consider the actual translation, which is chosen switching on the Axiom.AxiomType. The individual modal axioms are processed in separate functions (functions of the format eml_ProcessLogicX() in table 6.18). Each function returns a new TERM representing the translation of the modal axiom for each particular Def TERM (that is, member of the current instantiation set beginning with box). The TERM Def is processed as described in tables ? and ?. Subcomponents of the required output TERM are built up gradually, and then combined into the final TERM that is returned from the function. Variables required in the translation as are retrieved from the variable store (for example, eml_GetVar(VARX)). Where new predicate TERMs of the kind QDef need to be created during the translation, then the new symbols are created (or retrieved from the symbol cache) using the function eml_CreateAxiomSymbol(), supplying Def as the TERM on which to base the name of the new predicate. The modality index (for example, r in [](r,p)) is incorporated into predicate TERMs of the format $R_r(x,y)$ by calling the function eml_AxiomaticTranslationRel(), with the modality index obtained from the SYMBOL Axiom.Modality. For axioms 4^κ and 5^κ κ-factors (retrieved from Axiom.parameter1) that are outside the range 2-3 raise an error and translation stops. Likewise for axiom $alt_1^{\kappa1\kappa2}$, factors κ1 and κ2 (extracted from Axiom.parameter1 and Axiom.parameter2) that are outside the range 1–2 are not allowed, and for axiom B^κ only κ-factors in the range 2-3 are allowed. Extensive use is made of utility functions like eml_Negate() and eml_QuantifyTerm() during building the output TERMS. In many cases, special processing is needed for the TERM false passed as Def (see section ?). In some cases a new symbol is created during the processing of a modal axiom, for example for axiom D, a TERM of the format $\Box\neg\phi$ needs to be Defined. In order to achieve this the function eml_DefineNewDefs() is called from within the function eml_ProcessLogicX().

Table 6.18. Implementation of Translation of Axiom for the input problem

LIST **eml_AxiomLogicControl**(AXIOMDEFLIST *DEFs, FLAGSTORE, PRECEDENCE)
LIST **eml_ProcessAxiomLogic**(AXIOM Axiom, AXIOMDEFLIST *DEFs, FLAGSTORE, PRECEDENCE)
LIST **eml_ProcessLogic**(AXIOM Axiom, AXIOMDEFLIST *DEFs, FLAGSTORE, PRECEDENCE, TERM (*ProcessLogic)(AXIOM, TERM, AXIOMDEFLIST *, FLAGSTORE, PRECEDENCE))
TERM **eml_ProcessClassicalLogic**(AXIOM, FLAGSTORE, PRECEDENCE)
TERM **eml_ProcessLogicT**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicB**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicD**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogic4**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogic5**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicAlt1**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE,PRECEDENCE) TERM **eml_ProcessLogic4K**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE,PRECEDENCE) TERM **eml_ProcessLogic5K**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE,PRECEDENCE) TERM **eml_ProcessLogicAlt1KK**(AXIOM,TERM Def, AXIOMDEFLIST *,FLAGSTORE,PRECEDENCE)
TERM **eml_ProcessLogicSR**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicG**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicDEN**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicTR**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicM**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicW**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicBK**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicCR**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicCR2**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicCR3**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE) TERM **eml_ProcessLogicDBBB**(AXIOM,TERM Def,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE)
TERM **eml_ProcessPreviousComposition**(AXIOM,AXIOMDEFLIST *,FLAGSTORE, PRECEDENCE)

void **eml_UpdateCompositionList**(TERM Term, AXIOMDEFLIST *DEFs)
void **eml_UpdateCompositionAxiomLogic**(AXIOM Axiom,AXIOMDEFLIST *DEFs,FLAGSTORE,PRECEDENCE)
TERM **eml_QuantifyTerm**(SYMBOL Quantifier, TERM Term, LIST VarListContents)
void **eml_DefineNewDefs**(TERM,AXIOMDEFLIST*,LIST *OutTerms,FALGSTORE,PRECEDENCE,BOOL recursive)

Classical Translation of modal-axioms:

There are many predefined classical translations (correspondence properties) of modal axioms available to the user (correspondence properties for modal axioms T, D, 4, 5, B, alt_1, 5^κ, 4^κ and $alt_1^{\kappa 1,\kappa 2}$). They are defined in `eml_ProcessClassicalLogic()`, where a `switch` statement is used to select the correct processing for a particular axiom (switching on the constants `T, D, FOUR, FIVE, B, ALT1, 5K, 4K` and `ALT1KK`). When the correspondence property is to be incorporated into the axiomatic translation, then the `Def TERM` (indeed the entire instantiation set) is *not* required. Again, the `TERM` corresponding to the correspondence property is built by creating subcomponents and then building the final `TERM` from these fragments. When required, the function `eml_AxiomaticTranslationRel()` is used to create an $R_r(x,y)$ predicate (for example ??), with the modality index obtained from the `SYMBOL Axiom.Modality`. For axioms 4^κ and 5^κ, the κ factors (retrieved from `Axiom.parameter1`) is restricted to the range 2-3 (values outside this range raise an error and the translation stops). Likewise for axiom $alt_1^{\kappa 1,\kappa 2}$ both k factors (retrieved from `Axiom.parameter1 and Axiom.parameter2`) are restricted to the range 1-2 ($(\kappa 1,\kappa 2) = \{(1,1), (1,2), (2,1), (2,2)\}$).

When a correspondence property of a modal axiom is required that is not predefined in the software, there are two options. First the translation may be supplied in its entirety by the user, and protected from translation by the `noAxiom()` syntax. This is processed in the functions `eml_ProcessExempt()` and `eml_ProcessExemptHelper()`. The input `TERM` is passed to these functions, stripped of the `noAxiom()` formulae, and then returned. These exempt formula(e) are held in the `TERM ExemptFormula`, and re-introduced into the result of translation at the end of the controlling function `eml_Axiomatic()`. Second, if the axiom can be formulated as a modal formula of the type $\Diamond^h\Box^i p \rightarrow \Box^j\Diamond^k p$, then the `slf()` syntax may be used. This is accessed via the façade function `eml_ProcessSLF()`. The `slf()` syntax stripped from the input `TERM` in a similar fashion to that just described. The indices (h,i,j,k) of the modal operators are extracted from the formula input in the `slf()` syntax in the function `eml_ProcessSLFHelper()`, and passed to the function `eml_SLFtoCorrespondence()` where the `TERM` representing the correspondence property is generated, built piecemeal according to the numerical values of these indices. The result is held in the `TERM SLFformula`, and reintroduced into the translated formulae in the same way as just described. An error is raised if the input formula enclosed within the `slf()` syntax is of the wrong format, and the translation is terminated.

Table 6.19. Implementation of the translation of modal axioms.

AXIOM **eml_GetAxiom**(int Index)
BOOL **eml_SetAxiom**(AXIOM)
void **eml_BuildAxiom**(AXIOMTYPE, AXIOMMODE, int, int)
void **eml_BuildAxiomAt**(int, AXIOMTYPE, AXIOMMODE, int, int)
void **AxiomComposition**[Index]
TERM **eml_SLFtoCorrespondence**(SYMBOL R,int h,int i,int j,int k,FLAGSTORE,PRECEDENCE)
TERM **eml_ProcessSLFHelper**(TERM,TERM *SLFformula,BOOL *FoundSLF,int h,int i,int j,int k,FLAGSTORE, PRECEDENCE)
TERM **eml_ProcessSLF**(TERM,TERM *SLFformula,FLAGSTORE, PRECEDENCE)
TERM **eml_ProcessExemptHelper**(TERM,TERM *ExemptFormula)
TERM **eml_ProcessExempt**(TERM,TERM *ExemptFormula)
BOOL **eml_AxiomsAreClassical**()
BOOL **eml_AreAxiomsDefined**()
BOOL **eml_IsNewAxiom**(AXIOMTYPE)
void **eml_Count_Modalities**()

Program logic for translation of axioms: Strict order and Composition.

In the case of an axiomatic translation of a modal axiom, the program flow is as follows, for each axiom in turn, as defined in the axiom cache (details of the functions are given later):

1. **`eml_ProcessPreviousComposition()`** is called, and the result incorporated into the translation as stage 4.

2i. For axioms T, 4, B, D, alt_1, 4^κ, $alt_1^{\kappa\kappa}$, **`eml_ProcessLogic()`** called with the processing function relevant to the modal axioms, and the result stored for return at stage 4.

2ii. For axioms 5 and 5^κ, and axioms returning true from the function `eml_IsNewAxiom()` (that is, `SR, G, DEN, TR, M, H, BK, CR, CR2, CR3`)

(a) **`eml_UpdateCompositionAxiomLogic()`** is called.

(b) **`eml_ProcessPreviousCompoistion()`** is called, and the result stored for use in stage 4.

(c) **`eml_ProcessLogic()`** called with the relevant processing function, and result stored to use at stage 4.

3. For modal axioms not 5, nor 5^κ, nor returning true from the function `eml_IsNewAxiom()`, **`eml_UpdateCompositionAxiomLogic()`** is called

4. The results from 1, 2 and 3 are gathered and returned.

Details of the functions follow:

In *`eml_ProcessLogic()`* the `boxed` instantiation set `TERMs` are gathered from `DEFs.BoxDef` and `DEFs.XtraBoxDef`. The program flow loops through each of these, first checking that the modality index of the box of the instantiation set element matches the `Axiom.Modality` (that is, multi-modal *'editing'* on modality index is implemented here), and then calling the actual translation, using the axiom translation function previously selected in `eml_ProcessAxiomLogic()` (see the descriptions above for `eml_ProcessLogicX()`), finally collecting the `TERMs` produced at each iteration for return.

In *`eml_UpdateCompositionAxiomLogic()`* the compositional `TERMs` arising from the translation of the modal axiom are generated. The program flow is:

1. The correct processing is selected by `switch`ing on the `Axiom.AxiomType` (defined for *all* axioms).
2. `HiddenOpt.DoNotUseAxiomXComposition` can act as a guard statement terminating the procedure.
3. The compositional `TERMs` are built up from the members of the current instantiation set (`DEFs.BoxDef` + `DEFs.XtraBoxDef`), for example, composition for axiom D is $\Box\neg\phi$.
4. The result is returned via the function `eml_UpdateCompositionList()` called for each compositional `TERM`.

In *`eml_UpdateCompositionList()`* a `TERM` is screened, and stored temporarily in `DEFs.CompositionList`, provided that it is not already present in `DEFs.BoxDef` nor `DEFs.XtraBoxDefs` and `DEFs.CompositionList`. The origin of the input `TERM` is compositional `TERMs` taken from `eml_UpdateCompositionAxiomLogic()`.

In *`eml_ProcessPreviousComposition()`* the compositional (box) `TERMs` (nominally) from the *previous* modal axiom are processed. The function loops through the members of `DEFs.CompositionList`, using `eml_DefineNewDefs()` to DEFine these compositional `TERMs`, and gather the results in a `LIST` for return. At the end of this function `DEFs.CompositionList` is emptied by setting it equal to `list_Nil()`, with the contents being first transferred to `DEFs.XtraBoxDef` (thus updating the current instantiation set). `HiddenOpt.DoNotUseComposition` can act as a guard statement preventing access to the function.

In *`eml_DefineNewDefs()`*, new `TERMs` that have been used elsewhere in the translation are defined using `eml_DEF()`. The newly defined `TERM` is also processed recursively, selecting only those subcomponents being with a *box* `SYMBOL`. In each case, the newly defined `TERMs` are collected and returned in a LIST pointer. There is much potential for duplication, so all `TERMs` are tested with `eml_ListContains()` for duplication against `DEFs.BoxDef` and `DEFs.XtraBoxDef` before the new DEFinition takes place.

When an axiom is translated as a correspondence property, the logic is different. The correspondence property is simply added to the `LIST` that will form the '`FINAL TERM`'. Importantly, compositional `TERMs` that have been held-over from processing of the previous modal axiom are *not* processed. Instead, their processing is further delayed, are held over to be processed in the context of another modal axiom defined for axiomatic translation.

Illustration of the Algorithm for uni-modal case:

Since, the logic for the processing of a series of modal axioms is complicated, it is informative to consider an example. First consider three lists; (note: lists1 and 2 form the *current* instantiation set)

1. $\Box\chi^{\in}_{\varphi}$ - the list of $\Box$ subformulae of the input formula φ (the instantiation set) - never updated (`DEFs.BoxDEF`)

2. $\Box\beta$ – the list of extra $\Box$ TERMs that are now part of the instantiation set (updated from $\Box\gamma$ as appropriate; DEFs.XtraBoxDEF)
3. $\Box\gamma$ – the temporary list of $\Box$ TERMs held over from the previous cycle (usually from the previous modal axiom; DEFs.CompositionList).
Then consider six functions.
The first three add TERMs to the translated output ($\Box\phi$ is member of instantiation set beginning with box symbol).

- $Ax^A(\Box\phi)$ – the axiomatic translation of $\Box\phi$ for axiom *A*
- $Def_E(\Box\phi)$ – create new defintions in Def(Sf($\Box\phi$)), but only where $\Box\phi \notin (\Box\chi^{\in}_{\varphi} \cup \Box\beta)$
- $Ax^A_c()$ – create of a new correspondence property TERM for axiom *A* (see table below)

The second group of functions produce new box TERMs according to the formulae in the table immediately below.

- $NT^A_1(\Box\phi)$ – introduce new TERMs from translation of modal axiom, that *only* need to be DEFined
- $NT^A_2(\Box\phi)$ – introduce new TERMs from translation of modal axiom, similar to composition, added to the instantiation set
- $NT^A_3(\Box\phi)$ – introduce new TERMs from composition in translation of modal axiom, added to the instantiation set

Axiom	Ax()	$Ax_c()$	NT_1	NT_2	NT_3
T	$\forall x(\neg Q_{\Box r,p}(x) \vee Q_{r,p}(x))$	$\forall x(R(x,x))$	–	–	–
D	$\forall x(\neg Q_{\Box r,p}(x) \vee Q_{\neg\Box r,\neg p}(x))$	$\forall x \exists y(R(x,y))$	$\Box r,\neg p$	–	$\Box r,\neg p$
4	$\forall x(\neg Q_{\Box r,p}(x) \vee \forall y(\neg R(x,y) \vee Q_{\Box r,p}(y)))$	$\forall xy(R(x,y) \rightarrow R(y,x))$	–	–	$\Box r,\Box r,p$
5	$\forall x(\forall y(\neg R(x,y) \vee \neg Q_{\Box r,p}(y)) \vee Q_{\Box r,p}(x))$	$\forall xyz((R(x,y) \wedge R(y,z)) \rightarrow R(x,z))$	–	$\Box r,\neg\Box r,p$	$\Box r,\neg\Box r,p$
B	$\forall x(\forall y(\neg R(x,y) \vee \neg Q_{\Box r,p}(y)) \vee Q_{r,p}(x))$	$\forall xyz(R(x,y) \wedge R(x,z) \rightarrow R(y,z))$	–	–	$\Box r,\neg\Box r,p$
alt_1	$\forall x(\neg Q_{\neg\Box r,p}(x) \vee Q_{\Box r,\neg p}(x))$	$\forall xyz(R(x,y) \wedge R(x,z) \rightarrow (y \approx z))$	$\Box r,\neg p$	–	$\Box r,\neg p$
4^K	$\forall x(\neg Q_{\Box r,p}(x) \vee \forall y(\neg R^K(x,y) \vee Q_{\Box r,p}(y)))$	$\forall xy(R^{K+1}(x,y) \rightarrow R(x,y))$	–	–	$\Box^K r,\Box r,p$
5^K	$\forall x(Q_{\Box r,p}(x) \vee \forall y(\neg R^K(x,y) \vee \neg Q_{\Box r,p}(y)))$	$\forall xyz(R^K(x,y) \wedge R(x,z) \rightarrow R(y,z))$	–	$\Box^K r,\neg\Box r,p$	$\Box^K r,\Box r,\neg p$
alt_1^{KK}	$\forall xyz(\neg R^{K1}(x,y) \vee \neg R^{K2}(x,z) \vee \neg Q_{\neg\Box r,p}(y) \vee Q_{\Box r,\neg p}(z))$	$\forall xyz(R^{K1+1}(x,y) \wedge R^{K2+1}(x,z) \rightarrow (y \approx z))$	$\Box r,\neg p$	–	$\Box^{K2} r,\Box r,\neg p$ & $\Box^{K1} r,\Box r,p$

Now the algorithm for the translation of the modal axiom series $\alpha_1\alpha_2\alpha_3\ldots\alpha_i$ for a given formula is:

loopfor each modal axiom α_i
step 1 for α_i: if α_i is not correspondence property,
add translated TERMs to the output … $Def_E(\Box\gamma_{i-1})$
then update list … $\Box\beta_i = (\Box\beta_{i-1} \cup \Box\gamma_{i-1})$
then update list … $\Box\gamma_i$ = empty
else
nothing
step 2 for α_i: if α_i is not correspondence property
add translated TERMs to the output … $Ax^{\alpha i}(\Box\chi^{\in}_{\varphi} + \Box\beta_i) + Def_E(NT^{\alpha i}{}_1(\Box\chi^{\in}_{\varphi} + \Box\beta_i))$
$+ Ax^{\alpha i}(NT^{\alpha i}{}_2(\Box\chi^{\in}_{\varphi} + \Box\beta_i)) + Def_E(NT^{\alpha i}{}_2(\Box\chi^{\in}_{\varphi} + \Box\beta_i))$
else
add translated TERMs $Ax_c^{\alpha i}()$
step 3 for α_i: if α_i is not correspondence property and not axiomatic translation *without* composition
update list $\gamma_i = NT^{\alpha i}{}_3(\Box\chi^{\in}_{\varphi} + \Box\beta_i)$
else
nothing

For a very artificial example, modal formula $\varphi = \Box p$ in $KT_oD_o5_oD_cB_o4^*5_o$, the TERMs created at each step in the alogirithm, for each axiom, are illustrated below:

Here 4* is the translation of axiom 4 without composition

i^{th} axiom	**end of Step**	**new TERMs**	**$\Box\chi^{\in}_{\varphi}$**	**$\Box\beta$**	**$\Box\gamma$**
T_o or T	1	-	$\Box p$	-	-
	2	$Ax^T(\Box p)$	$\Box p$	-	-
	3	-	$\Box p$	-	-
D_o	1	-	$\Box p$	-	-
	2	$Ax^D(\Box p)$ + $Def(\Box\neg p)$	$\Box p$	-	-
	3	-	$\Box p$	-	$\Box\neg p$
5_o	1	$Def(\Box\neg p)$*	$\Box p$	$\Box\neg p$	-
	2	$Ax^5(\Box p, \Box\neg p)$, $Ax^5(\Box\neg\Box p, \Box\neg\Box\neg p)$, $Def(\Box\neg\Box p, \Box\neg\Box\neg p)$	$\Box p$	$\Box\neg p$	-
	3	-	$\Box p$	$\Box\neg p$	$\Box\neg\Box p, \Box\neg\Box\neg p$
D_c	1	-	$\Box p$	$\Box\neg p$	$\Box\neg\Box p, \Box\neg\Box\neg p$
	2	$Ax_c^D()$	$\Box p$	$\Box\neg p$	$\Box\neg\Box p, \Box\neg\Box\neg p$
	3	-	$\Box p$	$\Box\neg p$	$\Box\neg\Box p, \Box\neg\Box\neg p$
B_o	1	$Def(\Box\neg\Box p, \Box\neg\Box\neg p)$*	$\Box p$	$\Box\neg p$, $\Box\neg\Box p, \Box\neg\Box\neg p$	-
	2	$Ax^B(\Box p, \Box\neg p$, $\Box\neg\Box p, \Box\neg\Box\neg p)$	$\Box p$	$\Box\neg p$, $\Box\neg\Box p, \Box\neg\Box\neg p$	-
	3	-	$\Box p$	$\Box\neg p$, $\Box\neg\Box p, \Box\neg\Box\neg p$	$\Box\neg\Box p$, $\Box\neg\Box\neg\Box p$, $\Box\neg\Box\neg\Box\neg p$
4	1	$Def(\Box\neg\Box p^*, \Box\neg\Box\neg p^*, \Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p)$	$\Box p$	$\Box\neg p$, $\Box\neg\Box p$, $\Box\neg\Box\neg p$, $\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p$	-
	2	$Ax^4(\Box p, \Box\neg p$, $\Box\neg\Box p, \Box\neg\Box\neg p, \Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p)$	$\Box p$	$\Box\neg p$, $\Box\neg\Box p$, $\Box\neg\Box\neg p$, $\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p$	-
	3	-	$\Box p$	$\Box\neg p$, $\Box\neg\Box p$, $\Box\neg\Box\neg p$, $\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p$	- **
5_o or 5	1	-	$\Box p$	$\Box\neg p$, $\Box\neg\Box p$, $\Box\neg\Box\neg p$, $\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p$	-
	2	$Ax^5(\Box p, \Box\neg p$, $\Box\neg\Box p, \Box\neg\Box\neg p, \Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p)$ $Ax^5(\Box\neg\Box p, \Box\neg\Box\neg p, \Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p, \Box\neg\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg\Box\neg p)$ $Def(\Box\neg\Box p, \Box\neg\Box\neg p, \Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p, \Box\neg\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg\Box\neg p)$	$\Box p$	$\Box\neg p$, $\Box\neg\Box p$, $\Box\neg\Box\neg p$, $\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p$	-
	3	-	$\Box p$	$\Box\neg p$, $\Box\neg\Box p$, $\Box\neg\Box\neg p$, $\Box\neg\Box\neg\Box p, \Box\neg\Box\neg\Box\neg p$	- ***

* refers to added TERMs that are not needed because they already exist in the Defintions set.
**this is not $\Box\Box p$ because the axiom is not 4_o
***since this list is not processed, then it need not be updated

The distribution of these elements of functionality in the code can be seen by examining the previous section. It is apparent that the functionality of $Ax^A(\Box\phi)$ is defined mainly in eml_ProcessLogicX(), $Def_E(\Box\phi)$ in eml_eml_DefineNewDefs(), and $Ax^A_c()$ in eml_ProcessClassicalLogic(). However, in the case of the functions $NT^A_1(\Box\phi)$, $NT^A_2(\Box\phi)$, and $NT^A_3(\Box\phi)$, the functionality is spread between, for example, eml_UpdateCompositionalLogic() and eml_ProcessProviousComposition(), and it is the *order* in which these functions are called for a particular modal axiom that defines whether it corresponds to $NT^A_1()$, $NT^A_2()$ or $NT^A_3()$.

Notes on the Special Handling of True/False in the Translation:

It is possible for formula including the false ($\perp$) symbol to be submitted to and processed by extended SPASS. In severalcases special handling, not previously defined in other sections must be used. The key relationships required to handle these cases are (i) $\pi(\perp, x) = \perp$ which means $Q_\perp(x) = \perp$, and (ii) $\pi(\neg\perp, x) = \neg\pi(\perp, x) = \neg\perp$ which means $Q_{\neg\perp}(x) = \neg\perp$.

Special handing is required in for example, eml_BuildExistentialTerm() where an input $\perp$ returns $\perp$ ($\exists x.Q_P(x)$). Likewise, the handling of $\perp$ in eml_SimpleDEF() gives $\forall x(\perp \rightarrow \neg Q_{\neg\perp}(x))$. This case, and others handled by eml_SimpleDEF() and eml_DEF() are developed below.

$Def(\perp) = \forall x(Q_\perp(x) \rightarrow \pi(\perp,x)) \wedge \forall x(Q_\perp(x) \rightarrow \neg Q_{\neg\perp}(x)) \wedge \forall x(Q_{\neg\perp}(x) \rightarrow \pi(\sim\perp,x))$
$= \forall x(Q_\perp(x) \rightarrow \pi(\perp,x)) \wedge \forall x(Q_\perp(x) \rightarrow \neg Q_{\neg\perp}(x)) \wedge \forall x(Q_{\neg\perp}(x) \rightarrow \pi(\neg\perp,x))$
$= \forall x(Q_\perp(x) \rightarrow \perp) \wedge \forall x(Q_\perp(x) \rightarrow \neg Q_{\neg\perp}(x)) \wedge \forall x(Q_{\neg\perp}(x) \rightarrow \neg\pi(\perp,x))$
$= \forall x(Q_\perp(x) \rightarrow \perp) \wedge \forall x(Q_\perp(x) \rightarrow \neg Q_{\neg\perp}(x)) \wedge \forall x(Q_{\neg\perp}(x) \rightarrow \neg\perp)$
$= (\perp \rightarrow \perp) \wedge \forall x(Q_\perp(x) \rightarrow \neg Q_{\neg\perp}(x)) \wedge (\neg\perp \rightarrow \neg\perp)$
$= \top \wedge \forall x(Q_\perp(x) \rightarrow \neg Q_{\neg\perp}(x)) \wedge \top$
$= \forall x(Q_\perp(x) \rightarrow \neg Q_{\neg\perp}(x))$
$= \forall x(\perp \rightarrow \neg Q_{\neg\perp}(x))$

$Def(\neg\perp) = \forall x(Q_{\neg\perp}(x) \rightarrow \pi(\neg\perp,x)) \wedge \forall x(Q_{\neg\perp}(x) \rightarrow \neg Q_{\neg\neg\perp}(x)) \wedge \forall x(Q_{\neg\neg\perp}(x) \rightarrow \pi(\sim\neg\perp,x))$
$= \forall x(Q_{\neg\perp}(x) \rightarrow \neg\pi(\perp,x)) \wedge \forall x(Q_{\neg\perp}(x) \rightarrow \neg Q_\perp(x)) \wedge \forall x(Q_\perp(x) \rightarrow \pi(\perp,x))$

$= (\neg\bot \rightarrow \neg\bot) \wedge\ \forall x(Q_{\neg\bot}(x) \rightarrow \neg\bot) \wedge\ (\bot \rightarrow \bot)$

$= \forall x(Q_{\neg\bot}(x) \rightarrow \neg\bot)$

$Def(\Box\bot) = Def(\Box\bot) \wedge \forall x(Q_{\Box\bot}(x) \rightarrow \pi(\Box\bot,x)) \wedge\ \forall x(Q_{\Box\bot}(x) \rightarrow \neg Q_{\neg\Box\bot}(x)) \wedge\ \forall x(Q_{\neg\Box\bot}(x) \rightarrow \pi(\sim\Box\bot,x))$

$= Def(\Box\bot) \wedge\ \forall x(Q_{\Box\bot}(x) \rightarrow \pi(\Box\bot,x)) \wedge\ \forall x(Q_{\Box\bot}(x) \rightarrow \neg Q_{\neg\Box\bot}(x)) \wedge\ \forall x(Q_{\neg\Box\bot}(x) \rightarrow \pi(\neg\Box\bot,x))$

$= Def(\Box\bot) \wedge \forall x(Q_{\Box\bot}(x) \rightarrow \forall y(R(x,y) \rightarrow Q_{\bot}(y))) \wedge \forall x(Q_{\Box\bot}(x) \rightarrow \neg Q_{\neg\Box\bot}(x)) \wedge\ \forall x(Q_{\neg\Box\bot}(x) \rightarrow \exists y(R(x,y) \wedge Q_{\neg\bot}(y)))$

$= Def(\Box\bot) \wedge \forall x(Q_{\Box\bot}(x) \rightarrow \forall y(R(x,y) \rightarrow \bot)) \wedge \forall x(Q_{\Box\bot}(x) \rightarrow \neg Q_{\neg\Box\bot}(x)) \wedge\ \forall x(Q_{\neg\Box\bot}(x) \rightarrow \exists y(R(x,y) \wedge Q_{\neg\bot}(y)))$

A minority of the axiomatic translations of modal axioms also require special handling for TERMs involving $\bot$. These are listed below, for the input $\Box\bot$.

- axiom T : $\forall x(\neg Q_{\Box\bot}(x) \vee \bot)$
- axiom B : $\forall x(\neg R(x,y) \vee\ \neg Q_{\Box\bot}(y) \vee \bot)$
- axiom B^2 : $\forall x(\forall y(\neg R(x,y) \vee \forall z(\neg R(y,z) \vee \neg Q_{\Box\bot}(z))) \vee \bot)$
- axiom B^3 : $\forall x(\forall y(\neg R(x,y) \vee \forall z(\neg R(y,z) \vee \forall u(\neg R(z,u) \vee \neg Q_{\Box\bot}(u)))) \vee \bot)$
- axiom W : $\forall x(\exists y(R(x,y) \wedge Q_{\Box\bot}(y) \wedge \neg\bot) \vee Q_{\Box\bot}(x))$
- axiom SR : $\forall x \forall y(\neg R(x,y)\ \vee \neg Q_{\Box\bot}(y)\ \vee \bot)$
- axiom TR : $\forall x((\neg Q_{\Box\bot}(x)\ \vee \bot) \wedge (\neg\bot\ \vee Q_{\Box\bot}(x)))$

Modal axioms D, 4, 5, alt_1, 4^κ, 5^κ, $alt_1^{\kappa\kappa}$, G, M, DEN, CR, CR2, CR3, DBBB do not require special handling for $\bot$.

6.10. Pass collected Terms from tasks 7 & 8 to SPASS:

Throughout the translation process, TERMs are produced and accumulated in a LIST and various TERMs inside the function eml_Axiomatic(). At the end of eml_Axiomatic(), this TERM forms the 'FINAL TERM' that is passed to the SPASS resolution prover.

One of the final tasks before passing the translated 'FINAL TERM' to SPASS is to reset the precedence of the propositional symbols added during the translation. The entry point for this functionality is eml_SetPrecedence(), which takes the 'FINAL TERM' as one of the formal parameters. The parameter HiddenOpt.DoNotUseSetPrecedence can block entry to this function. eml_SetPrecedence() first populates the fields NegBox and AccessCount for each entry in the symbol cache (eml_AxiomSymbolList[]). The field NegBox (populated by functions eml_BuildNegBox(), eml_BuildNegBoxHelper(), and eml_BubbleAnd() using the TERM taken from AXIOMSYMBOL.Term of each entry in the symbol cache) contains a recursive analysis of the parent TERM for the symbol, containing a sorted, unique list of the $\Box$ and $\neg$ subcomponents. The field AccessCount (populated by the function eml_NoOfQs()) contains a count of the occurrence of each symbol in the 'FINAL TERM'. This information is then used to sort the symbols (using list_MergeSort() with a variety of comparator functions – see below), and then precedence for the symbols that were *newly created* within the eml-module, is reset according to this order using the function symbol_SetOrdering(). The predicate symbols are sorted in the order:

1. Separate lists are built for terms of the type eml_AXIOMREL (R) and eml_AXIOM (Q), which are put back together later in the order: accessibility relation symbols (R) > other predicate symbols (Q).
2. The number of occurrences of each predicate symbol in the translated 'final term' is assigned the next highest priority in ordering (with the least frequently occurring symbol assigned the highest precedence – using the function eml_CompareByCountPlus()).

The predicate symbols are further ordered (at a progressively lower priority, accessed via eml_CompareByBoxNot(), eml_CompareTermList() / eml_CompareTermListHelper()).

3. Q-terms containing 'and' terms > Q-terms without 'and' terms (using eml_ContainsAnd()).
4. deep Q-terms like $Q\Box\Box\Box_p$ > shallow Q-terms like $Q\Box\Box_p$ (using eml_NoOfBoxes() which counts the number of boxes in a term).
5. $Q_p > Q_q$ (see function eml_TerminalPredicate() - which uses a String comparison on (for example) p and q based upon the C function strcmp()).
6. $Q_{\neg Term} > Q_{Term}$ (using eml_NotIndex() - which returns the depth of the i^{th} $\neg$ symbol in a term).

The entry point into SPASS top.c needs to gather information of the precedence mode that has been set by the eml-module. This is done in the function eml_GetPredenceMode(). This information is used to ensure that if precedence has been set by the eml-module, it will be honored, except in the case that the set_precedence() syntax is used in the dfg file.

Finally, just before the 'final term' is passed to the SPASS resolution prover, the symbol table is printed to provide feedback to the user (eml_PrintSymbolTable()). The 'final term' is passed via eml_??().

Table 6.20. Passing collected Terms from Axiomatic Translation to SPASS

void **eml_SetPrecedence**(PRECEDENCE,TERM FinalTerm)
TERM **eml_BubbleAnd**(TERM,BOOL *Changed)
LIST **eml_BuildNegBoxHelper**(TERM)
void **eml_NoOfBoxes**(TERM,int *Count)
int **eml_NoOfQs**(TERM,SYMBOL *Symbol)
TERM **eml_TerminalPredicate**(TERM)
int **eml_NotIndex**(TERM,int Index,int *BoxDepth,int *NotCount)
void **eml_ContainsAnd**(TERM,BOOL *Contains)
int **eml_CompareTermListHelper**(TERM Left,TERM Right)
BOOL **eml_CompareTermList**(POINTER Ptr1,POINTER Ptr2)
void **eml_BuildNegBox**(AXIOMSYMBOL *,PRECEDENCE)
BOOL **eml_CompareByBoxNot**(POINTER Ptr1,POINTER Ptr2)
BOOL **eml_CompareByCountPlus**(POINTER Ptr1,POINTER Ptr2)
void **eml_PrintSymbolTable**()
int **eml_GetPredenceMode**()

6.11. Clean up memory allocated to data, etc in EML module:

Little cleanup needs to be done at exit from the axiomatic translation. Cleanup code is found in the function eml_AxiomCleanUp(), called from eml_Free() at the end of top.c.

Table 6.21. Implementation of clean up of memory in EML module

void **eml_AxiomCleanUp**()

6.12: The User Interface.

The code for the user interface is available in the school's file systems at P06/smithk/public_html (in sub-directories cgi-bin, KJSpass, mspass_opInfo), and can be downloaded at the project web site. The prototype code is implemented in a form suitable for viewing in the Firefox browser. Screen shots are seen in section ?. The code is implemented in XHTML, CSS, and JavaScript with some Ajax.

The parent page of the application is a static web page main.html. All the other screens are implemented by modification of the displayed elements in main.html, or addition of other static pages to main.html, which then acts as an enclosing framework. The JavaScript definitions for main.html are found in spass.js, and the CSS stylesheet definitions in template.css. The structure of the static html file is a series of XHTML definitions, using a great many DIV subdivisions, each identified by a specific id. Many of the elements defined in this structure are, by default, hidden from display, and changes in the structure of the page that is displayed within the browser are largely made by displaying and hiding components, as appropriate, using JavaScript to drive the changes. The XML tree of HTML elements is structured in a way that mirrors the visual grouping of displayed elements (top bar of tabs, left-hand sidebar of tabs, right-hand buttons bar, central work area, etc). The elements whose display status is toggled by JavaScript functions form successive sub-trees of these structures (for example the central work area DIV contains divisions corresponding to the main script central work area, the main results central work area, the parameters central areas, etc). Images form the leaf element of many of the branches of the XML-tree. The artwork for these is found in the sub-directory colors (as .png or .gif files). The buttons around the central window are implemented as hyperlinks with associated JavaScript actions. The Main Script work area is formed by an editable textarea. At startup it displays the same MSPASS file as was seen in the web interface of MSPASS at http://www.cs.man.ac.uk/schmidt-bin/web-mspass.cgi. The options area of Main is an INPUT element. iframes have been used to implement static pages displayed within the main.html framework. The main results work area is an iframe with source resultSet.html; the parameter area is an iframe with source parameters1.html, the modal parameters uses modal.html, the sample scripts

work area uses samplescript.html, the help area uses an iframe with source http://moodle.cs.man.ac.uk/mscwiki/index.php/Media_Handling (for demonstration purposes only).

The stylesheet in template.css uses an absolute layout model, defining the display properties of both named classes of elements and of individual elements. It is of fairly standard format, and delivers a fixed size webpage that resists all attempts to resize it. For this reason, it is best to use the default font set for display in the Firefox browser.

The JavaScript driving the interface is in spass.js, with a small amount of JavaScript code in the headers of the static html files. The JavaScript is structured as a series of interlinked and (where possible) reusable functions. The organization of the functions mirrors the organized layout of the displayable elements from the XML tree in main.html. In this way, actions and changes can be easily associated with coherent groups of elements. In general, elements of the main.html document are addressed by their ID. Frequently all of click, double-click and mouse-over functionality is defined for these elements. The elements of main.html which are displayed at any particular time (in response to mouse clicks made by the user), are defined by display modes or states. Hence, for example, when the Sample Scripts screen is displayed, all the elements of main.html that have variable visibility are hidden, and then the correct subset of elements of the top tab-bar, right-side tab-bar, left-side button-bar and workarea is displayed, by a cascade of functions associated with the display mode DisplaySampleScript. These display modes also store state information, so for example, upon returning to the Main Screen, if the Results sub-screen was previously displayed when the Main screen was last exited, it will be automatically displayed again. Since all the elements of the application interface co-exist in the same displayed page at one time (whether visible or hidden) it is easy to pass information between these elements.

Interaction with the web server cgi-bin is required in order to execute SPASS (called via

```
http://www2.cs.man.ac.uk/smithk-bin/spass0.cgi/?
+input_textArea=+escape(document.getElementById(mainScriptTextarea).value)+&
+=input_text=+escape(document.getElementById(mainScriptCommandLine).value))
```

and to add line numbers to the input file that is displayed along with the results from SPASS (via http://www2.cs.man.ac.uk/smithk-bin/lineNumbers.cgi/?…). The text data returned from these calls is displayed by direct replacement of the source (src) of an iframe element (here frame[0], the mainResultWorkArea), by directly manipulating the XML structure of the iframe *within* the browser. This method is of accessing HTML elements is used elsewhere in the code. The ordering of frames in main.html is [0] mainResultWorkArea, [1] parameterAreaStd1Frame, [2] parameterAreaModalFrame, [3] sampleScriptWorkAreaFrame, [4] historyAreaFrame, [5] helpAreaFrame. This is a relatively new and poorly documented method of manipulating HTML elements, and requires the use of a modern browser with (often) some trial-and-error to probe the structure of the HTML elements within the internal XML model that the browser forms from the displayed web page (a more difficult example is found in the function sampleScriptsCopyButtonAction() where text is extracted from a nested iframe containing a plain text source

```
var currentSampleContents =
window.frames[2].frames[0].document.getElementsByTagName(pre)[0].textContent;  )
```

A small amount of Ajax is used to display tooltips for the add history [+] and Sample Scripts (Sc) tabs. This code is adapted from Negrino & Smith[?]. The clock is a simple JavaScript function that updates the text content of a DIV element (again adapted from [?]). In order to add a new results/options/dfg script combinations to the history store, the JavaScript code creates HTML elements on-the-fly (by manipulating an XML subtree with DIV, PRE and A elements), and then appending them to the HTML elements displayed in the History display mode. Simple JavaScript objects are used here to organize the code that implements this mechanism.

The file samplescript.html uses a drop down menu system that is activated by mouse-over events [adapted from ?], and here the JavaScript is included in the header for the html file. The sample scripts themselves are held as plain text .dfg files in the sub-directory sampleScripts. The file parameters1.html provides the user with easy access to the standard SPASS options [?]. The structure of the code is an HTML table, that is used to organize input radio elements, and select elements. The actions associated with these elements are defined in spass.js, and the default selections correspond to the default values for these parameters in SPASS. A similar arrangement is seen for the axiomatic translation modal parameters include in modal.html. The JavaScript code is aware of the default selections, and is able to update the command line in the Main/Script screen with only those selected options that differ from these default selections. The code here makes use of JavaScript sub-functions, and regular expression and string processing. The processing of the modal parameters allows groups of flags to be defined by a single mouse-click, or allows the user to set at a fine grained level all the individual bits in the new modal options flags. The Results sub-screen of the Main screen

uses the iframe resultSet.html to organize the Results output and the line-numbered Script content. resultSet.html in turn organizes these two sets of information using an old-fashioned frameset. The sources of these two frames is directly manipulated by the JavaScript code (see above) in order to display updated information.

120 example SPASS .dfg scripts are found in the sub-directory mspass_opInfo/mspass/template. They were used during testing of the code written during this project. More example SPASS scripts (from the student accessible /opt/info/courses/CS616/mspass/dfg directory on the school's file system) are found in the sub-directory mspass_opInfo/mspass/dfg. They have been reorganized, and both sets are accessed via the contents.html, that appears by pressing the More Info button on the Sample Scripts screen.

The cgi-bin directory contains the compiled SPASS binary. The format of the binary should be suitable for the web server on which the cgi-script executes. SPASS executes from a simple Perl script – spass0.cgi. This is a greatly simplified version of the script that runs the web interface of MSPASS: Web Interactive Input Form at http://www.cs.man.ac.uk/schmidt-bin/web-mspass.cgi. It uses standard Perl cgi libraries, contains a simple pattern matching mechanism to help catch unauthorized input from hackers, and separates out the SPASS options collected in the argument (taken from the options area in the Main/Script screen, see above) by simple text processing, gathers the .dfg input from the argument (from the main editor in the Main/Script screen), writing it as a text file in a temporary location, then submits both these data to the SPASS binary, and returns the result (via an intermediate temporary file) to the calling webpage by printing plain text. LineNumbers.cgi uses a similar mechanism to add line number to the script text by executing /usr/bin/nl –ba.

6.13 Object Oriented Design in Java.

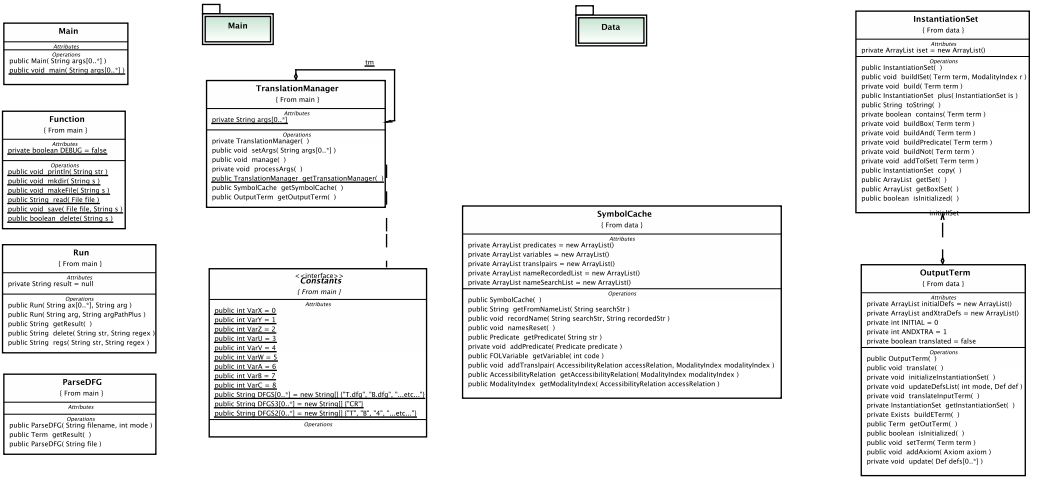
A duplicate Axiomatic Translation system (named Jasper) was built in Java and formulated as a test system. The code will compile on Java 1.4+, and uses no library functions more specialized than java.util.ArrayList. This system is intended for use only during testing, and hence many of the functions were developed to a less sophisticated level than for the C-implemention.

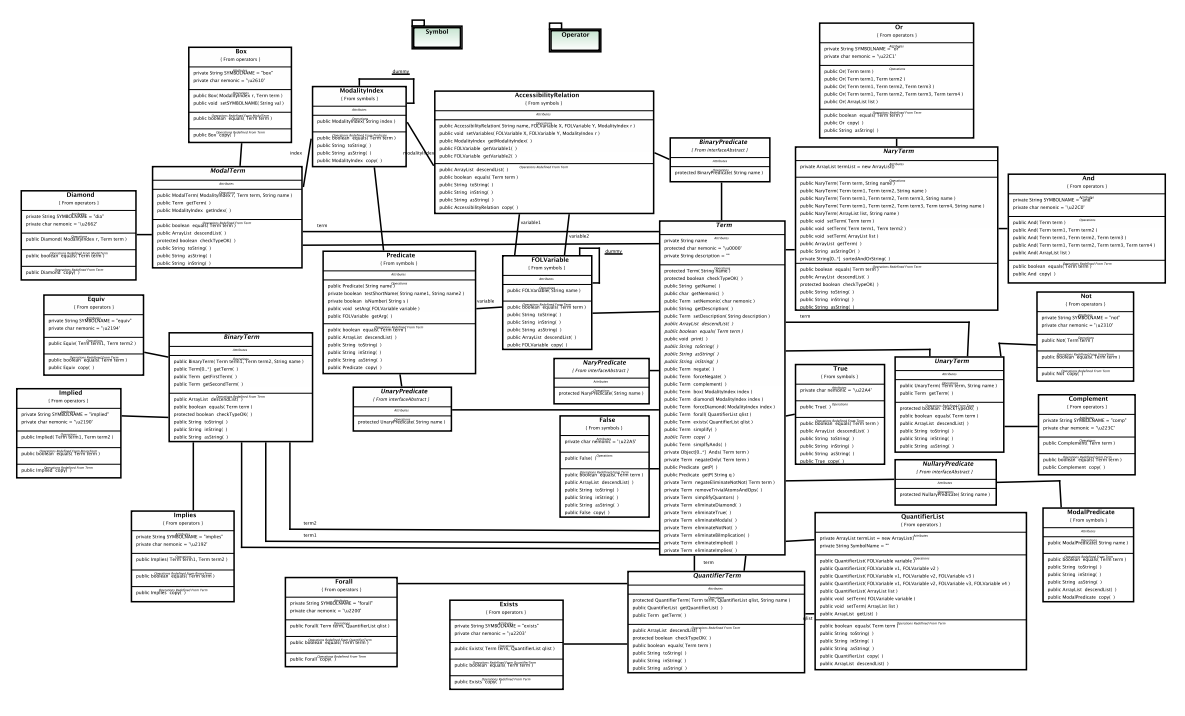
A simple UML class diagram of the principle components of the system is shown in figure 6.1. On examination, the design is seen to be simpler than for the C-implementation. Briefly, first it is necessary to provide a data structure that can store and manipulate SPASS-like terms. This is done very well using a hierarchy of classes extending the class Term, and ultimately providing a Term corresponding to every operator and symbol that is being considered (And, Forall, ModalPredicate, etc), and organizing functionality with abstract classes where appropriate (for example NaryTerm as the superclass of And and Or classes). Functionality such as copying, generating a String representation, testing for equality, etc, are easily distributed to each subclass of Term. Second, these terms need to be manipulated to form Definitions (Def object), or by Axiom objects (AxiomT, Axiom5, etc). The range of Axioms implemented is axiom 4, 4^2, 4^3, 5, 5^2, 5^3, alt_1, alt_1^{11}, alt_1^{12}, alt_1^{21}, alt_1^{22}, B, B^2, B^3, D, DBBB, Den G, M, SR, T, Tr, W. The SymbolCache, InstantiationSet and OutputTerm objects provide various utilities that actually perform the translation process and gather the translated terms (in a similar way to that seen in the C-code). To complete the Java translation, a control structure is added in the object TranslationManager (implementing the interface Constants), which defines the problems to be translated and the way in which they are to be translated (for example, translation using composition, or in a mode analogous to the default translation for Axiom 5, or the particular modality index to which an axiom is to be applied, in a similar way to that previously described for the C-implementation). One important difference between the object-oriented design and the C-implementation is that in the latter case, the translation is accumulated piecemeal during the operation of the software. In Japser, the terms requiring transltion are accumulated (for example, those requiring definition, or processing by a particular axiom), but are only actually translated at the end of the process. This is because objects are created for each case (for example, a Def object), and providing the translation is simply a method provided by the object.

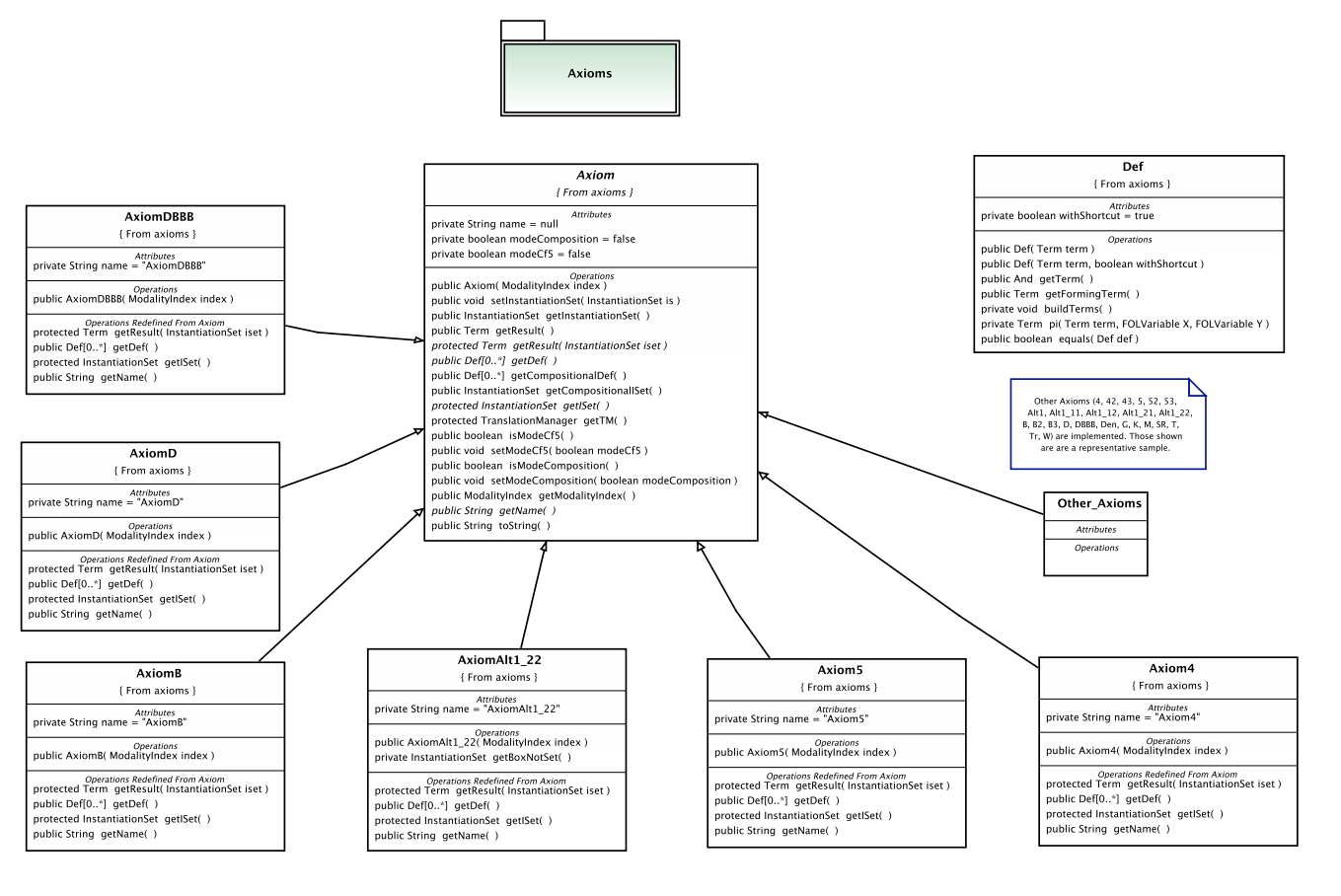
For use as a test system, the following approach was adopted. A particular problem is translated by the Java system and the translation temporarily stored (as a hierarchy of inter-related Term sub-objects). The TranslationManager also directs Java to call (via java.lang.Process in Run.java [following xx]) SPASS with appropriate axiom command-line switches and a pre-formulated .dfg file as arguments, in such a way that SPASS performs the *same* translation of the *same* problem. The TimeLimit switch is set at 2 seconds, effectively requesting SPASS to terminate as quickly as possible as soon as the translation was finished (so frequently no result was calculated). The "FINAL TERM" reported by the SPASS Axiomatic Translation system is the term that is passed into the resolution prover, and this is captured as a text item from the SPASS output (using elements of

the java.util.regex package developed into a grep-like function [following xx]). The text corresponding to this final term was parsed into a Java Term-based data structure (using a simple JavaCC parser [] controlled from ParseDFG.java, the command script for which can be found in parse/dfg.jj) and also temporarily stored. Finally the terms produced by these two independent translations are compared directly sub-term for sub-term, and determined to be equal or not. A record is made individually for each tested translation in the log files produced (in directories logs, logs2, logs3), and in the output printed to standard output by Java. If a single translation event in Java differs from the corresponding translation in SPASS, then the entire test fails. Clearly it was essential that the translation in Java be designed to produce terms (for example the formulation of each modal axiom) in the same format as the SPASS translation.

It is worth noting several points about this implementation. First, the detail or specificity of the sub-terms used in Jasper is greater than the TERMs and SYMBOLs in SPASS. Hence, modality indices like r (ModalityIndex), or modal predicates like p or q (ModalPredicate), or accessibility relations like R (AccesibilityRelation), etc, are represented by dedicated sub-classes of Term. Second, Jasper implements the axomatic translation of both uni-modal and multi-modal problems, but not bi-modal axioms. Likewise, correspondence properties are not avaialbe as translations for modal axioms. Finally, the range of possible SPASS terms that can be parsed by code derived from parse/dfg.jj is much smaller than that generally available in the definition of the dfg-syntax for SPASS. Hence, modal predicates can only take the values {“p”, “q”, “t”}; modality indexes can only take the values {“r”, ”s”, ”a”}; first order variables are restricted to the values {“U”-“Z”}. This is a limitation of the parser function that was developed for comparison with the translation produced by SPASS (see dfg.jj), and not a limitation of the code implementing the axiomatic translation in Java.


Main
Attributes
Operations
public Main(String args[0..*])
public void main(String args[0..*])
Function
{ From main }
Attributes
private boolean DEBUG = false
Operations
public void println(String str)
public void mkdir(String s)
public void makefile(String s)
public String read(File file)
public void save(File file, String s)
public boolean delete(String s)
Run
{ From main }
Attributes
private String result = null
Operations
public Run(String ax[0..*], String arg)
public Run(String arg, String argPathPlus)
public String getResult( )
public String delete(String str, String regex)
public String regs(String str, String regex)
ParseDFG
{ From main }
Attributes
Operations
public ParseDFG(String filename, int mode)
public Term getResult( )
public ParseDFG(String file)
Main
1m
TranslationManager
[From main]
Attributes
private String args[0..*]
Operations
private TranslationManager( )
public void setArgs(String args[0..*])
public void manage( )
private void processArgs( )
public TranslationManager getTranzastionManager( )
public SymbolCache getSymbolCache( )
public OutputTerm getOutputTerm( )
<<interface>>
Constants
{ From main }
Attributes
public int VarX = 0
public int VarY = 1
public int VarZ = 2
public int VarU = 3
public int VarV = 4
public int VarW = 5
public int VarA = 6
public int VarB = 7
public int VarC = 8
public String DFGS[0..*] = new String[] {"T.dfg", "B.dfg", "...etc..."}
public String DFGS3[0..*] = new String[] {"C'&"}
public String DFGS2[0..*] = new String[] {"T'", "B", "A", "...etc..."}
Operations
Data
SymbolCache
{ From data }
Attributes
private ArrayList predicates = new ArrayList()
private ArrayList variables = new ArrayList()
private ArrayList transIpairs = new ArrayList()
private ArrayList nameRecordedList = new ArrayList()
private ArrayList nameSearchList = new ArrayList()
Operations
public SymbolCache( )
public String getFromNameList(String searchStr)
public void recordName(String searchStr, String recordedStr)
public void namesReset( )
public Predicate getPredicate(String str)
private void addPredicate(Predicate predicate)
public FOLVariable getVariable(int code)
public void addTransIpair(AccessibilityRelation accessRelation, ModalityIndex modalityIndex)
public AccessibilityRelation getAccessibilityRelation(ModalityIndex modalityIndex)
public ModalityIndex getModalityIndex(AccessibilityRelation accessRelation)
InstantiationSet
{ From data }
Attributes
private ArrayList iset = new ArrayList()
Operations
public InstantiationSet( )
public void buildISet(Term term, ModalityIndex r)
private void build(Term term)
public InstantiationSet plus(InstantiationSet is)
public String toString( )
private boolean contains(Term term)
private void buildBox(Term term)
private void buildAnd(Term term)
private void buildPredicate(Term term)
private void buildNot(Term term)
private void addToISet(Term term)
public InstantiationSet copy( )
public ArrayList getISet( )
public ArrayList getBoxISet( )
public boolean isInitialized( )
InstantiationSet
OutputTerm
{ From data }
Attributes
private ArrayList initialDefs = new ArrayList()
private ArrayList andXtraDefs = new ArrayList()
private int INITIAL = 0
private int ANDXTRA = 1
private boolean translated = false
Operations
public OutputTerm( )
public void translate( )
private void initializeInstantiationSet( )
private void updateDefsList(int mode, Def def)
private void translateInputTerm( )
private InstantiationSet getInstantiationSet( )
private Exists buildETerm( )
public Term getOutTerm( )
public boolean isInitialized( )
public void setTerm(Term term)
public void addAxiom(Axiom axiom)
private void update(Def defs[0..*])


Symbol
Operator
Box
{ From operators }
private String SYMBOLNAME = "box"
private char mnemonic = "\u2610"
public Box(Mode MODEINDEX, ... Term term)
public void setSYMBOLNAME(String val)
public boolean equals(Term term)
public Box clone()
ModalityIndex
{ From symbols }
public ModalityIndex(String index)
public ModalityIndex clone()
public String toString()
public String asString()
public ModalityIndex copy()
Access
Index
ModalityIndex
AccessibilityRelation
{ From symbols }
public AccessibilityRelation(String name, FOLVariable X, FOLVariable Y, ModalityIndex r)
public void setVariableX(FOLVariable X, FOLVariable Y, ModalityIndex r)
public ModalityIndex getModalityIndex()
public FOLVariable getVariableX()
public FOLVariable getVariableZ()
public ArrayList descendList()
public boolean equals(Term term)
public String toString()
public String asString()
public AccessibilityRelation copy()
ModalTerm
public ModalTerm(Modality MODALITY, Term term, String name)
public Term getTerm()
public ModalityIndex getIndex()
term
Diamond
{ From operators }
private String SYMBOLNAME = "dia"
private char mnemonic = "\u2662"
public Diamond(Modality box, ... Term term)
public boolean equals(Term term)
public Diamond clone()
Equiv
{ From operators }
private String SYMBOLNAME = "equiv"
private char mnemonic = "\u2194"
public Equiv(Term[] TERM, ... Term term2)
public boolean equals(Term term)
public Equiv copy()
Implied
{ From operators }
private String SYMBOLNAME = "implied"
private char mnemonic = "\u2192"
public Implied(TwiTerm term1, Term term2)
public boolean equals(Term term)
public Implied copy()
Implies
{ From operators }
private String SYMBOLNAME = "implies"
private char mnemonic = "\u2192"
public Implies(TwiTerm t, Term term2)
public boolean equals(Term term)
public Implies copy()
term2
term1
Forall
{ From operators }
private String SYMBOLNAME = "forall"
private char mnemonic = "\u2200"
public Forall(Term TERM, QuantifierList qlist)
public boolean equals(Term term)
public Forall copy()
Exists
{ From operators }
private String SYMBOLNAME = "exists"
private char mnemonic = "\u2203"
public Exists(Term TERM, QuantifierList qlist)
public boolean equals(Term term)
public Exists copy()
Predicate
{ From symbols }
public Predicate(String name)
private boolean isNumberString(String name1, String name2)
private boolean isNumber(String s)
public void setArgs(FOLVariable variable)
public FOLVariable getArg()
public boolean equals(Term term)
public ArrayList descendList()
public String toString()
public String asString()
public Predicate copy()
variable
FOLVariable
{ From symbols }
public FOLVariable(String name)
public boolean equals(Term term)
public String toString()
public String asString()
public ArrayList descendList()
public FOLVariable copy()
variable1
variable2
dummy
Term
private String name
protected char mnemonic = "\u0000"
private String description = ""
protected Term(String name)
protected boolean checkTypeOK()
public String getName()
public char getMnemonic()
public Term setMnemonic(char mnemonic)
public String getDescription()
public Term setDescription(String description)
public ArrayList descendList()
public boolean equals(Term term)
public void print()
public String toString()
public String asString()
public String toString()
public Term negate()
public Term forceNegate()
public Term complement()
public Term[] term(ModalityIndex index)
public Term diamond(ModalityIndex index)
public Term forceDiamond(ModalityIndex index)
public Term[] term(QuantifierList qlist)
public Term clone(QuantifierList qlist)
public Term simplify()
public Term copy()
public Term simplifyAnds()
private Object[][] findTerms(Term term)
private Term negateOne(Term term)
public Predicate getP()
public Predicate getP(String s)
public Term negateEliminateNotNot(Term term)
private Term removeTrivialAtomsAndOps()
private Term simplifyQuantors()
private Term eliminateDiamond()
private Term eliminateTrue()
private Term eliminateModals()
private Term eliminateNotNot()
private Term eliminateBiImplications()
private Term eliminateImplies()
private Term eliminateImplies()
BinaryPredicate
{ From interfaceAbstract }
Attributes
protected BinaryPredicate(String name)
NaryPredicate
{ From interfaceAbstract }
Attributes
protected NaryPredicate(String name)
UnaryPredicate
{ From interfaceAbstract }
Attributes
protected UnaryPredicate(String name)
BinaryTerm
public BinaryTerm(Term left, Term term2, String name)
public Term[] ...*() getTerm( )
public Term getFirstTerm( )
public Term getSecondTerm( )
public ArrayList descendList()
public boolean equals(Term term)
protected boolean checkTypeOK()
public String toString()
public String asString()
public String asString()
False
{ From symbols }
private char mnemonic = "\u2249"
public False()
public boolean equals(Term term)
public ArrayList descendList()
public String toString()
public String asString()
public String asString()
public False copy()
Or
{ From operators }
private String SYMBOLNAME = "or"
private char mnemonic = "\u2228"
Operations
public Or(Term term)
public Or(Term term1, Term term2)
public Or(Term term1, Term term2, Term term3)
public Or(Term term1, Term term2, Term term3, Term term4)
public Or(ArrayList list)
public boolean equals(Term term)
public Or copy()
public String asString()
NaryTerm
private ArrayList termList = new ArrayList()
Attributes
public NaryTerm(Term term, String name)
public NaryTerm(Term term1, Term term2, String name)
public NaryTerm(Term term1, Term term2, Term term3, String name)
public NaryTerm(Term term1, Term term2, Term term3, Term term4, String name)
public NaryTerm(ArrayList list, String name)
public void setTerm(Term term)
public void setTerm(Term term1, Term term2)
public void setTerm(ArrayList list)
public ArrayList getTerm()
public String asString()
private String[0, 1] sortedAndOrString( )
public boolean equals(Term term)
public ArrayList descendList()
protected boolean checkTypeOK()
public String toString()
public String asString()
public String asString()
term
And
{ From operators }
private String SYMBOLNAME = "and"
private char mnemonic = "\u2227"
Operations
public And(Term term)
public And(Term term1, Term term2)
public And(Term term1, Term term2, Term term3)
public And(Term term1, Term term2, Term term3, Term term4)
public And(ArrayList list)
public boolean equals(Term term)
public And copy()
Not
{ From operators }
private String SYMBOLNAME = "not"
private char mnemonic = "\u2310"
public Not(Term[] TERM)
public boolean equals(Term term)
public Not copy()
Complement
{ From operators }
private String SYMBOLNAME = "comp"
private char mnemonic = "\u223C"
public Complement(Term term)
public boolean equals(Term term)
public Complement copy( )
True
{ From symbols }
private char mnemonic = "\u2244"
public True()
public boolean equals(Term term)
public ArrayList descendList()
public String toString()
public String asString()
public String asString()
public True copy()
UnaryTerm
Attributes
public UnaryTerm(Term term, String name)
public Term getTerm( )
protected boolean checkTypeOK()
public boolean equals(Term term)
public ArrayList descendList()
public String toString()
public String asString()
public String asString()
NullaryPredicate
{ From interfaceAbstract }
Attributes
protected NullaryPredicate(String name)
QuantifierList
{ From operators }
private ArrayList termList = new ArrayList()
private String SymbolName = ""
Attributes
public QuantifierList(FOLVariable variable)
public QuantifierList(FOLVariable v1, FOLVariable v2)
public QuantifierList(FOLVariable v1, FOLVariable v2, FOLVariable v3)
public QuantifierList(FOLVariable v1, FOLVariable v2, FOLVariable v3, FOLVariable v4)
public QuantifierList(ArrayList list)
public void setTerm(FOLVariable variable)
public void setTerm(ArrayList list)
public ArrayList getList()
public boolean equals(Term term)
public String toString()
public String asString()
public String asString()
public QuantifierList copy( )
public ArrayList descendList( )
QuantifierTerm
Attributes
protected QuantifierTerm(Term term, QuantifierList qlist, String name)
public QuantifierList getQuantifierList( )
public Term getTerm( )
public ArrayList descendList()
protected boolean checkTypeOK()
public boolean equals(Term term)
public String toString()
public String asString()
public String asString()
qlist
term
ModalPredicate
{ From symbols }
public ModalPredicate(String name)
public ModalPredicate(FOLVariable term)
public String toString()
public String asString()
public ArrayList descendList()
public ModalPredicate copy()


Axioms
Axiom
{ From axioms }
Attributes
private String name = null
private boolean modeComposition = false
private boolean modeCf5 = false
Operations
public Axiom(ModalityIndex index)
public void setInstantiationSet(InstantiationSet is)
public InstantiationSet getInstantiationSet( )
public Term getResult( )
protected Term getResult(InstantiationSet iset)
public Def[0..*] getDef( )
protected InstantiationSet getISet( )
public String getName( )
public Def[0..*] getDef( )
public InstantiationSet getCompositionalISet( )
protected InstantiationSet getISet( )
protected TranslationManager getTM( )
public boolean isModeCf5( )
public void setModeCf5(boolean modeCf5)
public boolean isModeComposition( )
public void setModeComposition(boolean modeComposition)
public ModalityIndex getModalityIndex( )
public String getName( )
public String toString( )
AxiomDBBB
{ From axioms }
Attributes
private String name = "AxiomDBBB"
Operations
public AxiomDBBB(ModalityIndex index)
Operations Redefined From Axiom
protected Term getResult(InstantiationSet iset)
public Def[0..*] getDef( )
protected InstantiationSet getISet( )
public String getName( )
AxiomD
{ From axioms }
Attributes
private String name = "AxiomD"
Operations
public AxiomD(ModalityIndex index)
Operations Redefined From Axiom
protected Term getResult(InstantiationSet iset)
public Def[0..*] getDef( )
protected InstantiationSet getISet( )
public String getName( )
AxiomB
{ From axioms }
Attributes
private String name = "AxiomB"
Operations
public AxiomB(ModalityIndex index)
Operations Redefined From Axiom
protected Term getResult(InstantiationSet iset)
public Def[0..*] getDef( )
protected InstantiationSet getISet( )
public String getName( )
AxiomAlt1_22
{ From axioms }
Attributes
private String name = "AxiomAlt1_22"
Operations
public AxiomAlt1_22(ModalityIndex index)
private InstantiationSet getBoxNotSet( )
Operations Redefined From Axiom
protected Term getResult(InstantiationSet iset)
public Def[0..*] getDef( )
protected InstantiationSet getISet( )
public String getName( )
Axiom5
{ From axioms }
Attributes
private String name = "Axiom5"
Operations
public Axiom5(ModalityIndex index)
Operations Redefined From Axiom
protected Term getResult(InstantiationSet iset)
public Def[0..*] getDef( )
protected InstantiationSet getISet( )
public String getName( )
Axiom4
{ From axioms }
Attributes
private String name = "Axiom4"
Operations
public Axiom4(ModalityIndex index)
Operations Redefined From Axiom
protected Term getResult(InstantiationSet iset)
public Def[0..*] getDef( )
protected InstantiationSet getISet( )
public String getName( )
Def
{ From axioms }
Attributes
private boolean withShortcut = true
Operations
public Def(Term term)
public Def(Term term, boolean withShortcut)
public And getTerm( )
public Term getFormingTerm( )
private void buildTerms( )
private Term pi(Term term, FOLVariable X, FOLVariable Y)
public boolean equals(Def def)
Other Axioms (4, 42, 43, 5, 52, 53, Alt1, Alt1_11, Alt1_12, Alt1_21, Alt1_22, B, B2, B3, D, DBBB, Den, G, K, M, SR, T, Tr, W) are implemented. Those shown are are a representative sample.
Other_Axioms
Attributes
Operations