A LOGIC DATABASE TO SUPPORT CONFIGURATION MANAGEMENT IN Ada

34-82 1986

Patrizia Asirelli
Paola Inverardi
Istituto di Elaborazione della Informazione del C.N.R., Pisa, Italy.

1 *INTRODUCTION*

The Programming Language Ada (Ada 1983), directly faces aspects related to software production by introducing concepts like program library, modules and separate compilation issues. Modules management or, using a wider term, configuration, is intended as the activity of binding together different components into a system thus bringing, in the programming language, elements of programming in the large (DeRemer 1975). In Ada, modules are composed through *with* clauses that are simply interpreted as dependency relations among modules, with respect to compilation, recompilation and linking activities.

In order to be able to reason about configuration issues, it would be desirable to have a suitable environment to rigorously define the meaning of compilation, recompilation, linking etc., in terms of the components and the relations among them.

We propose a logic framework which permits both, to formalize such configurational activity (thus extending the benefits of a formal definition to the language supporting environment) and to introduce, in the programming environment, a very flexible tool to reason about modules properties and configuration strategies. We have to cope with an evolving world where the effects of activities, such as compilation and linking, are the creation and modification of relations among objects. This brings us to consider a database approach and, in particular, a Logic Database (LDB) one, to handle creation of new relations and objects.

An LDB approach is proposed with respect to traditional relational (or entity-relationship) data bases partly because of the capabilities that logic brings to relational databases (Gallaire et al.1984, Asirelli et al.1986), i.e. the ability to make deductions (deduce new facts from existing ones by means of rules), and also the ability to prove properties (integrity constraints) of the theory as logic program properties (Kowalski 1979).

Major achievements of the proposed logical approach are related to the capability of expressing very general constraints, and the quite straightforward way of defining advanced concepts, like histories and configurations, in terms of relations and properties of the objects in the database.

The proposed approach to configuration management rely on a prototype logic database management system, EDBLOG, which is based on a particular approach to Integrity Constraints checking (Asirelli et al. 1985), and implemented in the MPROLOG language on a IBM 3081, under VM.

*Work partially supported by Progetto Strategico Tecnologie dell'Informazione, Sottoprogetto Intelligenza Artificialé, C.N.R. Italy.

2 *LOGIC DATABASES AND EDBLOG*

A deductive database, defined as a set of facts (Extensional Components) and a set of rules (Intensional Components), can be seen as a first order theory, in particular, as a Horn clause theory. Thus, given the procedural interpretation of Horn clauses, a deductive database can be regarded as a logic program. Integrity constraints (properties which the database must posses), can be considered as properties of logic programs, thus assimilating the problem of integrity constraint checking to that of logic programs property proving. In addition, a deductive database system should offer much more than a logic programming system, since its objects are evolving first-order theories (databases), rather than a single fixed one. In the EDBLOG system a database is considered as a logic program plus a set of formulas expressing integrity constraints. Such formulas must be proved to be true in the minimal model of the program, thus ensuring the correctness of the data base with respect to the integrity constraints.

Hence a logic data base can be seen as:

1) *a set of facts*, the Extensional component of the DB (EDB), which are unit Horn clauses;
2) *a set of deductive rules*, the Intensional component of the DB (IDB), which are definite Horn clauses;
3) *a set of Integrity Constraints* (IC), which are formulas of the form:

   $A_k \rightarrow B_1, \ldots, B_s$

   whose informal interpretation is that whenever A_k is true then B_1 and...and B_s must also be true;
4) *a set of control formulas* are either formulas as in 3) or else

   i) $A_1 \wedge \ldots A_m \rightarrow B_1, \ldots, B_n$

   ii) $\rightarrow B_1, \ldots, B_n$

   iii) $A_1 \wedge \ldots \wedge A_m \rightarrow$

The informal interpretation for i) is that whenever A_1 and ... and A_m are true then B_1 and ... and B_n must also be true; analogously ii) means that B_1 and ... and B_n must be true and, finally, iii) means that A_1 and ... and A_m must be false.

The two forms of *Integrity Constraints* and *Controls* formulas are used by EDBLOG in two different ways. IC are used to modify facts and rules of the data base so that only those facts which satisfy IC will be derivable from the data base theory, i.e. the semantics of the resulting DB is given by all facts that can be deduced from the data base and which satisfy IC. The second form of constraints, the controls, are used periodically (at user request), to check that changes to the data base have preserved consistency with respect to this sort of constraints.

Characteristic of this system is the possibility of modifying the data base, by adding or deleting facts, rules, IC and Controls, thus allowing to deal with evolving theories. The original system (DBLOG) has been further extended by adding a new theory in order to deal with transactions, i.e. compound updating operations thus resulting in the actual system EDBLOG. The language to express transactions syntactically resembles Concurrent Prolog, with no annotated variables. It allows to express transactions as follows:

$$\mathit{trans} \leftarrow \mathbf{prec_1} \mid \mathit{trans}_1, \ldots, \mathit{trans}_n \mid \mathbf{post_1}$$

$$\mathit{trans} \leftarrow \mathbf{prec_2} \mid \mathit{trans}_1, \ldots, \mathit{trans}_n \mid \mathbf{post_2}$$

The informal interpretation is that, in order to execute the operation *trans*, it is necessary to first verify which precondition ($\mathbf{prec_1}$ or $\mathbf{prec_2}$) holds, and then commit to the clause whose precondition has succeeded, executing the body and verifying the corresponding postcondition. The commit operation is, as for Concurrent Prolog a way of expressing the behaviour of the Prolog cut operator.

Preconditions and postconditions in transaction definitions will operate as a particular form of controls which must be checked before/after the execution of that particular set of operations (body of the transaction). Since checking for consistency in a DB can be very heavy and time consuming preconditions and postconditions are introduced to separate controls which are global to the DB, from those which are related to the particular transaction, thus reducing the number of global controls.

The operational interpretation of these transaction definitions is the standard Prolog resolution of clauses where clauses are tried in the order they appear in the program. Thus, the *commitment* will be to the first clause whose pre-condition part succeeds. The successful evaluation of a transaction causes the *Controls* formulas to be checked. The required transaction operation is aborted if this *Controls* checking fails. The abortion of a transaction is automatically handled (by backtracking), by ensuring that elementary updating operations are backtrackable upon failure.

3 *CONFIGURATION ISSUES*

Configuration refers to the activity that *links* smaller components into a system. This should be performed at different stages in the life cycle of a software system, with various objectives, i.e. it should be possible to express, within a uniform and consistent framework, managerial requirements and design decisions as well as any application language constraints, e.g. constraints that define compilation, recompilation and linking in terms of the components and the relations among them. LDBs allow to directly address configuration issues.

LDBs permits a theory of objects and their relations to be expressed, declaratively, thus providing a formal setting within which configurations can be examined and evaluated. On the other hand, the procedural semantics of Horn Clause Logic makes it feasible to validate the configurations , e.g. system specifications against user requirements, etc.

It is worth while stressing again that due to the procedural interpretation of HCL, the definition of a relation can be executed thus providing a uniform mechanism to *define* and to *run* configurations.

Configuration, in general, is an activity which might require expertise knowledge and it has been succesfully dealt with by espert systems. R1 (McDermott 1981) is a well known example of a hardware configuration expert system. The logic approach taken can be considered a first step towards the introduction of analogous techniques also in the field of software configuration.

An Ada program is defined as a collection of *compilation units* submitted to a compiler one or more times. Each compilation unit specifies the separate compilation of a construct which can be either a body part, a specification part or else a subunit. The compilation units of a program belong to a *program library* .

Dependencies among units are defined by *with clauses* , which allow a compilation unit to refer to other library units, thus achieving direct access to the units declared inside them. Dependencies are used to define, in the program library, a partial order among units to be taken into account when defining compilation, recompilation and execution activities.

To summarize, a program library consists of i) a collection of objects, the compiled units; ii) a set of relations among objects (the dependencies) such that certain conditions hold: e.g. library units must all have different names; for each secondary unit a corresponding library unit must exists, etc.

The flexibility of our approach allows the standard Ada *programming in the large* constructs to be also consistently extended, with new ways of putting modules together. For example, at the design level, the development of an application for a distributed target (Inverardi et al. 1985) may require a suitable environment for statically checking the adequacy of the application to be effectively mapped on a distributed machine (e.g. no modules with storage can be shared, etc.) This can be easily achieved by adding new relations and constraints to be verified.

3.1 *A logic database approach to configuration in Ada*

In this section we give a short sketch of the logic definition of the Ada Configuration Environment (ACE), while the complete description can be found in the Appendix.

The ACE, is described as a logic database, according to the described approach (Asirelli et al. 1985). The system is supposed to interact with the external environment (namely the editor, the compiler, etc.) to incrementally create and modify the theory by either adding or removing assertions (facts). The interaction with the external environment can easily be achieved provided that, the language (Prolog-like) used to implement the logic database has a primitive feature to call the operating system shell.

Facts correspond to the definition of the units in the theory. They contain all information about units which the editor is supposed to extract from the syntactic Ada code. *Rules* define the concepts of library unit, secondary unit and main program according to the Ada definition. *Constraints* are introduced in order to guarantee the consistency of the theory with respect to the global properties of the objects in the ACE, such as, for example, the uniqueness of library unit names. For example, it is possible to define:

facts

```
...
subp_decl (unit (a, null))
subp_body (unit (d, 1))
subp_body (unit (a, 1))
sub_unit (unit (fd, 1))
child_of (unit (d, 1), unit (fd, 1))
with_clause (unit (d, 1), ( unit (a, null)  . unit(b, null) . nil) )
...
```

Facts define four units ((a,null), (d,1), (a,1) and (fd,1)). Unary predicates denote the type of the units (subprogram declaration, subprogram bodies and a subunit), the argument is a data constructor representing the unit name by an identifier and its version number. Declarations have no version number, this is denoted by the constant **null**. The child_of predicate represent a dependency relation between units (d,1) and (fd,1), i.e. (fd,1) is a subunit of (d,1). Lastly, the with_clause predicate states for the unit (d,1) its list of referred library.

rules

```
....
main ( unit (X, Y)) ← subp_body ( unit ( X,Y ) )
lib_unit (X) ← decl (X)
lib_unit (X) ← main (X)
sec_unit (X) ←body (X)
sec_unit (X) ←sub_unit (X)
has_decl_part (unit(X,Y)) ← Y=/= "null", subp_decl (unit (X,null) )
....
```

The above rules assumes that the relations decl and body are also given, then they state the main property of a subprogram body; the library property of a main and of declaration units; the secondary unit property of bodies and sub_bodies. Furthermore, has_decl_part define the property of a unit (subprogram body) having a corresponding declaration unit.

Integrity Constraints

```
...
main(unit(X,Y))  →not (has_decl_part (unit(X, Y)) )
...
```

The above *IC*, will cause the rule defining the main relation to be modified. The resulting rule will thus state that, a main is a subprogram body with no corresponding declaration unit.

Controls

...

sub_unit (unit(X,Y)) $\rightarrow$ Y $\neq$ **"null"**

body (unit(X,Y)), **not** (main(unit(X,Y))) $\rightarrow$decl (unit(X,Y))

...

The *Controls* state that all subunits must be bodies and that there must exist a declaration unit for all bodies which are not main units.

An Ada Program Library relation is then defined. All the compiled units in the ACE are in the relation:

prog_lib (mylib,Y) $\leftarrow$ compiled (Y)

Therefore, the ACE also contains facts to state that a unit is compiled.

Since EDBLOG provides a transaction facility, it is possible to define the interaction with the compiler, as well as the interaction with all the other tools of the environment involved in modules management, as a transaction:

compile (X) $\leftarrow$sec_unit(X), compilable_sec(X)
| ***comp_Ada*** (X),**insert_f** (compiled_sec(X))

compile (X) $\leftarrow$ lib_unit(X), compilable_lib(X)
| ***comp_Ada*** (X), **insert_f** (compiled_lib(X))

Compile , is defined by two clauses, one for each type of units, library or secondary, respectively. The relations compilable_lib and compilable_sec act as pre-conditions that have to be verified by the database, before the commitment phase; ***comp_Ada*** is the call to the Ada compiler which is no more supposed to do any check on the compilability of the unit to be compiled, with respect to the PL structure; the **insert_f** operation, provided by EDBLOG, manipulates the database inserting a new fact in the theory.

Transactions guarantee to leave the theory unchanged upon any failure of the body, thus including the failure of the Ada compiler. Furthermore, the use of transactions consents to achieve a complete integration between the logic database system and the supporting environment.

Recompilation has been defined in the same style. A **remove_f** operation is used since the effects of recompiling a unit is, in general, the compilation of the unit itself while making obsolete the compilation of all those units which rely on its definition.

Once such framework has been set up, one can define, at the transaction level, different configuration strategies which can take into account the existence of different versions of the same unit depending on a supplementary piece of information (the *type* of the unit). We define a *configuration* as a set of *compiled* units that constitutes a complete program. In particular, a *Configuration* consists of:

i) a *main* compiled module, i.e. the module from which a complete program starts. The *main* module must be a *Library* of kind *Subprogram Body* for which a corresponding *Subprogram Declaration* module does not exist. this corresponds to the definition of main in Ada (Ada 1983).

ii) a set of compiled unit, associated to the *main* module by relations such as *with clause* and *sub_unit*.. In the case of subunits, each unit is a certain version while, in the case of *with_clauses* units are *Libraries* and thus they will not have versions.

iii) ricorsively, all compiled units, linked to all other modules found with relation *with clause*, *sub_unit* and *secondary _unit*, the last one denoting the body unit of a declaration one.

From the above definition one can easily see that the complexity of collecting modules together to make a configuration is quite high and the help of automatic mechanisms is almost essential .

In order to define a *configuration* , units must have the same type (last, default, etc).Thus, let us suppose that the database also contain, for each unit its type, e.g.

```
...
type (unit (a, 1), last).
type (unit(fg, 2), bad).
...
```

The configuration operation is defined as a set of transactions. In the following only the first clause of the necessary definitions is given:

config (A,TYPE) ← main (A), type(A,TYPE), compiled_lib(A), with(A,Y)
| *config_lib*(Y, TYPE, W), *config_sec* (A, TYPE, W1),
append (W, W1, CONF), **insert_f** (config(A,TYPE,CONF)).

Note that, the pre-conditions test that: there is a compiled main A with the given type TYPE and dependency list Y (the associated *with clause*). If the pre-conditions hold then a configuration CONF is started taking into account the *Library* list Y (*config_lib* is a transaction that builds a list of units, in the configuration, starting with the given list of library Y), and the *sub_units* (*config_sec*) of the main A. Configuration builds a list of units' names, in which the same unit can appear more than once, this is because, for semplicity, no clean up of the list has been made. It is anyway important to note that the list gives also an order which is, by construction, consistent with the partial ordering defined by the with_clauses. Thus, the configuration can serves as input for the linker as well as elaboration order.

4. *CONCLUSION*

The paper presents an approach to configuration management that relies on the use of a logic database as a project database, in an Ada programming environment.

Our claim is that the use of logic databases allows us to define, in a very simple and straightforward way the configuration facilities of Ada. In the appendix, the complete definition of a configuration environment for Ada is presented, which includes all the definitions stated by the Ada language (ref. chapter 10 of the RM) plus a proposal for versions managing and configuration control.

It is worthwhile to note that only few pages of declarative clauses were sufficient to specify all those concepts that generally represent a relevant part of the software needed in a Mapse.

REFERENCES

Ada (1983). Reference Manual for the Ada Programming Language, U.S.A. Dep. of Defense, ANSI/MIL-STD 1815 A, Juanary 1983.

Asirelli, P., De Santis, M., Martelli, M. (1985). Integrity Constraints in Logic Data Bases, *Journal of Logic Programming*, Vol. 2, n. 3, Oct. 1985, pp. 221-232.

Asirelli, P., et al.(1986), The Knowledge Base Approach in the Epsilon Project. In *ESPRIT'85: Status Report of Continuing Work*, The Commission of the European Communities (Eds.), Elsevier Science Pub. B. V., (North Holland), 1986.

DeRemer, F. & Kron, H. (1975). Programming in-the-large versus programming-in-the-small, *Proc. Conference Reliable Software*, pp.114-121, 1975.

Gallaire, H., Minker, J., Nicolas, J. (1984). Logic and Databases: a deductive approach, *Computing Surveys*, 16, (2), pp. 153-185, 1984.

Inverardi,P., Mazzanti, F., Montangero, C. (1985). The Use of Ada in The Design of Distributed Systems, *Proc. Ada International Conference*, Paris, May 1985.

Kowalski, R.A. (1979). *Logic for Problem Solving*, North Holland, Artificial Intelligence Series, N.J. Nilsson (Ed.), 1979.

Mc Dermott, J. (1981). R1 The Formative Years, *AI MAGAZINE*, Summer 1981.

APPENDIX

Ada Configuration Environment definition

Assertions (due to the interaction with the editor)

```
subp_decl (unit (a, null))
gen_decl (unit (c, null))
pack_decl (unit (b, null))
...
subp_body (unit (d, 1))
subp_body (unit (a, 1))
pack_body (unit (b, 1))
...
sub_unit (unit (fd, 1))
sub_unit (unit (fd, 2))
...
child_of (unit (d, 1), unit (fd, 1))
child_of (unit (d, 1), unit (fd, 2))
...
with_clause (unit (d, 1), ( unit (a, null) . unit(b, null) . nil) )
with_clause ( unit (a, null), ( unit (b, null) . nil) )
with_clause ( unit (c, null), ( unit (d, null) . nil) )
with_clause (unit (fd, 1), ( unit (h, null) . nil) )
with_clause (unit (fd, 2), ( unit (w, null) . nil) )
...
```

Assertions (due to the interaction with the compiler)

```
compiled_sec (unit (a, 1) )
compiled_lib (unit (a, null) )
compiled_lib (unit (d, 1) )
...
```

Rules (according to the Reference Manual)

```
decl (unit (X, null) ) ← subp_decl( unit (X, null ) )
decl ( unit (X, null ) ) ← gen_decl( unit (X, null ) )       -- declarative unit's definition
decl ( unit (X, null ) ) ← pack_decl( unit(X, null ) )

body (unit ( X, Y)) ←subp_body ( unit (X, Y) )
body ( unit ( X, Y)) ←pack_body ( unit (X, Y) )              -- body's definition
body ( unit ( X, Y)) ←gen_body ( unit (X, Y) )

main ( unit (X, Y)) ← subp_body ( unit ( X,Y ) )             -- main's definition
```

```
lib_unit (X) ← decl (X)
lib_unit (X) ← main (X)                              -- library's definition

sec_unit (X) ←body (X)
sec_unit (X) ←sub_unit (X)                           -- secondary's definition

has_decl_part (unit(X,Y)) ← Y=/= "null", subp_decl (unit (X,null) )

prog_lib(A,Y) ← compiled_list (Y)                    --Program Library's definition
...
```

Rules (stated for compilation)

```
compiled_lib_list (nil)
compiled_lib_list(X.Y)          ← compiled_lib(X), compiled_lib_list(Y)

compiled(X)                     ← compiled_lib(X)
compiled(X)                     ← compiled_sec(X)

compilable_lib(X)               ←  lib_unit(X), with_clause(X, Y), compiled_lib_list (Y)

compilable_sec(unit(X,Y))       ←  body (unit(X,Y)), compiled_lib (unit(X, null)),
                                    compiled_withs (unit (X,Y))
compilable_sec(unit(X,Y))       ←  sub_unit (unit(X, Y)),child_of (unit(Z,W), unit(X,Y)),
                                   compiled_sec(unit(Z,W)), compiled_withs (unit(X,Y))

compiled_withs(unit(X,Y))       ←  with_clause(unit(X,Y),Z), compiled_lib_list(Z)
```

Integrity Constraints (constraints on the main nature)

```
main(unit(X,Y))  →not (has_decl_part (unit(X, Y)) )
sec_unit (unit(X,Y)) →not (main (unit(X,Y)))
```

Constrols

```
subp_body (unit(X,Y)) → Y ≠ "null"
pack_body (unit(X,Y)) → Y ≠ "null"
```

sub_unit (unit(X,Y)) → Y ≠ "**null**"

body (unit(X,Y)), **not** (main(unit(X,Y))) →decl (unit(X,Y))

child_of(X,Y) →body(X), sec_ unit(Y)

Transactions (transactions realize a sort of tool level)

Compile : The external interaction with the compiler is defined as a transaction:

```
1.1  compile (X)          ←sec_unit(X), compilable_sec(X)
                           | comp_Ada (X),insert_f (compiled_sec(X))

1.2  compile (X)          ← lib_unit(X), compilable_lib(X)
                           |  comp_Ada (X), insert_f (compiled_lib(X))
```

insert_f is the insertion operation on the object theory (adds an assertion of kind compiled_...(a)). The definition leaves the user to specify what kind of unit should be compiled.

Only one transaction is necessary to define compilation; two clauses are needed in order to distinguish between library and secondary units, all the checks are performed by using the two rules compilable_...(X).

Recompile

```
1.1  recompile (X)   ←  sec_unit(X), compiled_sec(X)
                     |  remove_f(compiled_sec(X)), remove_child_of (X),compile (X).

1.2  recompile (X)   ←  lib_unit(X), compiled_lib(X)
                     |  remove_f(compiled_lib(X)), remove_lib (X),compile (X).

2.1  remove_child_of (X)  ←   child_of(X,Y), compiled_sec(Y)
                          |   remove_f(compiled_sec(Y)), remove_child_of (Y),
                              remove_child_of (X)

2.2  remove_child_of (X)  ←


3.1  remove_lib (X)        ←   main(X) | remove_child_of (X),remove_connect_lib (X),
                                         remove_connect_sec (X)

3.2  remove_lib (unit(X,null)) ←   body(unit(X,Y)), compiled_sec (unit(X,Y))
                               |   remove_f(compiled_sec(unit(X,Y))),
                                   remove_child_of ( unit(X,Y) ),
                                   remove_connect_lib (unit(X,null)),
                                   remove_connect_sec ( unit(X,null)
4.1  remove_connect_lib (X)   ←   compiled_lib(Y), with_clause(Y,Z),  member(X,Z)
                              |   remove_f (compiled_lib (Y)), remove_lib (Y),
                                  remove_connect_lib (X)

4.2  remove_connect_lib (X)   ←


5.1  remove_connect_sec (X)   ←   compiled_sec(Y),with_clause(Y, W), member(X,W)
                              |   remove_f(compiled_sec(Y)), remove_child_of (Y),
                                  remove_connect_sec (X)

5.2  remove_connect_sec (X) ←
```

remove_f is the remove operation on the object theory (removes an assertion of kind compiled_...(a)). Recompilation is defined in five transactions.

1. It is the main transaction, it removes the units from the program library together with all their dependent ones.
2. It removes all the subunits of a given unit together with their own subunits.
3. It removes all the dependent units of a library unit.
4. It removes all the library units which depend on X, i.e all the library units whose with list contains X.
5. It removes all the secondary units that depend on X.

Configurations

In order to define a *configuration* , units must have the same *type* (last, default, etc).Thus, the database also contain, for each unit its type, note that declarations cannot have types.

```
type (unit(a,1), last).
type (unit(fd, 2), bad).
...
```

```
1.1   config (A,TYPE) ← main (A), type(A,TYPE), compiled_lib(A), with(A,Y)
                      |  config_lib(Y, TYPE, W), config_sec (A, TYPE, W1),
                         append (W, W1, CONF), insert_f (config(A,TYPE,CONF))

1.2   config (A,TYPE) ← main (A), type(A,TYPE), compiled_lib(A),
                      |  config_sec (A, TYPE, CONF),
                         insert_f (config(A,TYPE,CONF))


2.1   config_lib (nil,TYPE, CONF)  ←

2.2   config_lib ( X1.REST, TYPE, X1.CONF)  ← with_clause(X1, W_list),
                                               compiled_lib(X1)
                                              | config_lib  (W_list, TYPE, CONF1),
                                              config_sec (X1, TYPE,CONF2),
                                              append ( CONF1, CONF2, CONF3),
                                              config_lib ( REST, TYPE, CONF4),
                                              append ( CONF3, CONF4, CONF)
```

```
3.1   config_sec (unit(X,null), TYPE, unit(X,Y). CONF) ← body(unit(X,Y)),
                              type(unit(X,Y),TYPE),compiled_sec(unit(X,Y)),
with_clause(unit(X,Y),W_list)
                         |  config_lib ( W_list , TYPE, CONF1),
                            config_sub (unit(X,Y), TYPE, CONF2),
                            append (CONF1, CONF2, CONF)

3.2   config_sec (unit(X,null), TYPE, unit(X,Y). CONF) ←
            body(unit(X,Y)), type(unit(X,Y),TYPE), compiled_sec(unit(X,Y))
                         |  config_sub (unit(X,Y), TYPE, CONF)

3.3   config_sec (unit(X,Y), TYPE, CONF) ←main(unit(X,Y))
                         |  config_sub (unit(X,Y), TYPE, CONF)



4   config_sub (X, TYPE, CONF) ←find_all(Y,child_of (X,Y),Z)
                         |  config_sub2(Z,TYPE, CONF),


5.1 config_sub2  (nil, TYPE, nil) ←

5.2 config_sub2(X.Y, TYPE, X.CONF) ← compiled_sec(X),
                                    type(X, TYPE), with_clause(X,W_list)
                         |  config_lib(W_list,TYPE,CONF1),
                            config_sub (X,TYPE, CONF2),
                            append(CONF1,CONF2,CONF3),
                            config_sub2(Y,TYPE,CONF4),
                            append(CONF3,CONF4,CONF)

5.3 config_sub2(X.Y, TYPE, X.CONF) ← compiled_sec(X), type(X, TYPE)
                         |  config_sub (X,TYPE, CONF1),
                            config_sub2(Y,TYPE,CONF2),
                            append(CONF1,CONF2,CONF)
```

Configuration has been realized with five transactions.

1. It is the main transaction, given a main and a TYPE it builds a list of units which realizes a configuration of the program starting from the given main and collecting only units of the given TYPE. Two transactions are needed since the case in which the main has no withs it must be taken into account. The reason for this is that we have assumed that an assertion with _clause(...) exists only if it is not an empty list. This assumptions allows us to reduce the number of facts in the data base but slightly complicates some transaction definitions, see also 3.

2. It configurates, starting from a list of library units (with_list). In some sence it allows us to configurate traversing the tree of dependencies in horizontal.

3. It allows us to configurate in vertical starting from a secondary unit and configuring all its subunits.

4. and 5. configurate all the subunits of a given unit.