A Logic Approach to Configuration in Ada

Patrizia Asirelli

Paola Inverardi

Istituto di Elaborazione della Informazione - CNR

Via S. Maria, 46

I56100 PISA

ITALY

Abstract

This paper presents a new application of logic programming in the area of integrated programming environments for algorithmic languages. Integrated programming environments are based upon the existence of a project database which is the repository of all information relevant to programming project throughout the life cycle of the project. In particular, modules management (configuration), is a typical activity which requires relationships among objects to be handled.

The paper addresses the advantages of using logic to express and manage configuration activities of languages which explicitely incorporate concepts related to software production. A logic database approach is proposed to support the configuration activity of the Ada programming language as an example of application of logic in the field of project data bases.

1. Introduction

New programming languages as Ada [Ada 83], ARGUS [Liskov 83], (C)Mesa [Mitchell 79], directly face aspects related to software production, 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 75].

Focussing on Ada, our aim is to formalize such configurational activity thus extending the benefits of a formal definition to the supporting environment of the language itself. Configuration, in fact, ranges over the programming language and its supporting environment [Hunke 81, AdaEnv 84].

While in principle the programming in the large methodology would allow to define general ways to putting components together to make systems, in practice, in Ada, it is intended as a supporting methodology to achieve separate compilation facilities. Thus, configuration facilities are defined a part from the proper language and their definition is deeply related to implementation issues.

In Ada the only way to compose modules together is through *with clauses* which are simply interpreted as dependency relations with respect to the compilation, linking and execution activities. Nevertheless, there is a need to formally express the setting in which such relationships among components exist. In other words, it would be desirable to have an environment which allows to rigorously define what compilation, recompilation, linking etc. mean, in terms of components and relations among them. Furthermore, such an environment would allow to extend the set of relations in order to define other properties of components [Inverardi 84] while ensuring compatibilities with the standard Ada configuration requirements.

Horn clause logic allows to express, in a declarative style, a theory of objects and their relations thus providing a suitable formal setting to reason about the standard Ada configuration environment and its possible extentions. On the other hand, the procedural semantics of Horn clause logic makes it feasible to validate the specifications against the user's request.

We want to model an evolving theory where the effects of activities such as compilation and linking create and modify relations among objects. This brings us to consider a data base approach and, in particular, a logic data base one to handle creation of new relations and objects.

The motivation of choosing a logic data base approach with respect to traditional relational (or entity-relationship) data bases is, besides the need for a formal setting, due to the extensions of capabilities that logic brings to relational data bases [Gallaire 84]. That is, the ability to make deductions (deduce new facts from old ones, by means of rules), and the ability to prove properties (integrity constraints) of the theory as logic program properties [Kowalski 79].

2. A Proposal For A Logic Environment for Ada

2.1 A logic data base approach

A deductive database, defined as a set of facts (Extensional Components) and a set of rules (Intensional Component), can be seen as a first order theory, in particular, as a Horn clause theory. A data base can be regarded as a Logic Program [Gallaire 84], thus Integrity Constraints (properties which the data base must possess), can be considered as properties of logic programs, assimilating the problem of integrity constraints checking to proving logic programs properties. In addition, a deductive database system should offer much more than a logic programming system does, since its objects are evolving first-order theories (databases), not a single fixed one. In particular, the problems of consinstency and redundancy should be faced.

Although logic programming offers a straightforward way of implementing deductive databases, some restrictions are needed to guarantee the termination of the query evaluation process and the evaluation of negative queries. Thus the class of logic programs has to be restricted to hierarchical program definitions which, in particular, do not allow recursive definitions [Clark 78, Shepherdson 84]. This restriction can be partially relaxed, at least with respect to negation and to certain kind of queries [Barbuti 85].

We will rely on a logic database management system DBLOG which has been sketched in [Asirelli 84, Asirelli 85a]. The approach considers a data base 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.

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:

$$A_k \rightarrow B_1 ,..., B_s$$

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) $A_1 \wedge ... \wedge A_m \quad \rightarrow B_1,...,B_n$

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.

Note that, for formulas i)-iii), as well as for formula 3), all variables are intended to be universally quantified, apart from local variables (i.e. variables occurring only in the right hand side) which are instead, intended to be existentially quantified.

The two forms of *Integrity Constraints* and *Controls* formulas are used by DBLOG 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 system can be seen as an amalgamated theory [Bowen 82, Furukawa 84] consisting of a meta-theory (the theory which handles the evolution of the data base), and the object theory (the logic data base).

The system can be further extended to provide a limited form of recursion [Barbuti 85] and by adding a new theory in order to deal with transactions, i.e. compound updating operations.

The language to express transactions syntactically resembles Concurrent Prolog, with no annotated variables [Shapiro 83]. 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 such transaction definitions is 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 precondition succeeds: *or-nondeterminism* is not achievable. Or-nondeterministic behaviour can be obtained by defining transactions which do not have the commitment operation, i.e. standard Prolog clauses.

Furthermore, providing that operations on DB are "backtrackable" (i.e. their effects are *undone* on failure), the effects of a transaction would be undone on a failure occurring either in the postcondition or in some operations of the body, thus obtaining an *and-nondeterministic* behaviour of the clauses.

2.2 Configuration Facilities in Ada

One of the main features of Ada, is that it makes the idea of incremental program development precise by its notion of program library and compilation. In fact, Ada provides, to the process of software design and development, its advanced linguistic constructs (packages, generics, tasks, etc) and separate compilation features (program libraries, subunits, etc).

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

Compilation units which are specification parts or *main* programs, are called *library units*, while body parts or subunits are called *secondary units.*

The compilation units of a program belong to a *program library* . The effect of compiling a compilation unit is to define or redifine (due to recompilation), this unit as a component of the 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 entities 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. libray units must all have different names; for each secondary unit there must be a corresponding library unit, etc.

Specifying these units and their relations as described above, is not straightforward as it can appear at a first glance. In fact, the existence of secondary units which can be structured in a tree fashion (subunits), depending on a root (either a main or a secondary unit), creates a set of exceptions to the general rules. For example, for a secondary unit which is a subunit, the constraint on the existence of a corresponding library unit does not apply (it applies only to secondary units which are roots). In the same way, the double role which a subprogram body can play, either as a main (if it is a library unit) or as a secondary unit, makes some relation definitions cumbersome.

2.3 A logic data base approach to configuration in Ada

In this section a logic definition of the Ada Configuration Environment, from now on ACE, is sketched. The detailed definition is given in the Appendix.

The ACE is described as a logic data base according to the approach introduced in section 2.1. The definition assumes the existence of an interactive external environment, namely the editor, the compiler, etc. whose role is that of modifying the theory either adding or removing assertions.

We assume an initial theory consisting of ground unit clauses (facts), rules and constraints.

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. This information is the minimum needed to define relations among units.

For example, the following two facts:

```
gen_decl (c,d.nil)
sub_unit (dfg,h.nil,f.d.nil)
```

define two units where, the predicate names are the type of the units (generic declaration unit and subunit), the arguments represent respectively: The name of the unit, the list of library units referred to by that unit and, for subunits only, the path to reach the subunit starting from the root unit. Note that the name of a subunit is obtained composing its simple name with the path.

Rules define the concepts of library unit, secondary unit and main program according to the Ada definition. For example:

main (X,Y) ← subp_body(X,Y)

lib_unit(X,Y) ← main (X,Y)

sec_unit(X,Y,nil) ← subp_body(X,Y)

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 the uniqueness of library unit names. For example:

sec_unit (X,Y,W), subp_body (X,Y) → lib_unit (X,Z), subp_decl (X,Z) states that, for all secondary units which are subprogram bodies, there must exist a library unit with the same name, which is a subprogram declaration. The above constraint does not prevent the insertion of a subprogram body as a library unit, provided that its declaration part does not already exist. This allows a main program to be a library unit while being a subprogram body.

An Ada Program Library (APL) is then defined as the set of compiled units in the ACE.

prog_lib (mylib,Y) ← compiled_list (Y)

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

We could assume, as for the editor interaction, that once a unit has been successfully compiled, a fact such as *compiled_lib (unitname)*, will be added to the theory. The insertion of the fact is successful if consistent with the rules which state the compilability of a unit. Thus besides the rules defining compilability of units, constraints exist which prevent from inconsistent insertions:

compiled_lib(X) → compilable_lib(X)

compiled_sec(X) → compilable_sec(X)

This approach allows to precisely specify configuration concepts, but a part from specification purpouses, it does not allow the language supporting environment to benefit of the executable nature of the specified theory. In order to achieve a better integration between the logic data base environment and the Ada supporting environment, the use of the Extended DBLOG (with transactions) is proposed. EDBLOG acts as a practical tool to be integrated in the Ada supporting environment, thus providing to the other tools all the functionalities related to modules management. In this framework it is possible to define the interaction with the compiler as a transaction:

compile (X,sec) ← compilable_sec(X) | ***comp_Ada*** (X), **add**(compiled_sec(X))

compile (X,lib) ← compilable_lib(X) | ***comp_Ada*** (X), **add**(compiled_lib(X))

Compile , is defined by two clauses, one for each type of units, library or secondary. This is necessary because, given the name of a unit, there is no other way of identifying the unit to be compiled as a library unit or as its secondary. Such problem has to be considered by any Ada environment since it arises from the fact that, in Ada, the same name is used to denote a specification part and its corresponding body part. This implicit relation causes no trouble from the language point of view, but it creates ambiguities once modelling the ACE. It can be solved either by some name convention or asking the user to explicitly supply the necessary information.

The two relations compilable_lib, compilable_sec act as preconditions that have to be verified before the commitment phase. Preconditions here take the role of the previously defined integrity constraints.
comp_Ada is a 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;
add inserts 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.

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

3. Conclusion

In this paper an approach to integrate a logic data base system with an Ada programming environment has been presented.

The aim was twofold: i) to provide a formal framework to specify the Ada configuration facilities, in order to formally define the interface between the the language and its programming environment; ii) to propose logic data bases as effective tools to be used in *traditional* programming environments.

Logic data bases offer great flexibility in handling an evolving world of objects and relations. This attitude makes LDBs suitable to deal with advanced programming environments supporting requirements like open-endness. In fact, the insertion of a new tool in the environment may require the specification of new relations among existing objects, and/or the creation of new objects. On the other hand, for effinciency purposes, a LDB can be easily implemented by accessing, at least for the facts, a traditional repository database while retaining relations and constraints control [Asirelli 85b]. In addition, the use of transactions allows to achieve a complete integration between the logic data base system and the supporting environment.

Focussing on our approach, in which a LDB has been proposed to handle modules management, flexibility allows to consistently extend the standard Ada *programming in the large* constructs, with new ways of putting modules together. For example, the development of an application for a distributed target [Inverardi 83] may require the environment to provide, at the design level, a tool to statically check the suitability of the application to be effectively mapped on a distributed machine (e.g. no shared modules with storage) [Inverardi 84]. We can easily achieve this goal by simply adding new relations and new constraints to be verified.

Configuration, in general, is an activity which might require expertise knowledge and it has been succesfully dealt with by espert systems. R1 [McDermott 81] is a well known example of a hardware configuration expert system. In this paper, configuration has been considered as that area, in programming environment, which can greatly benefit of the proposed logic approach. Furthermore the relations induced on modules by the configuration activity are typical examples of relations that have to be dealt with by a project data base.

References

[Ada 83] **Reference Manual for the Ada Programming Language**, U.S.A. Dep. of Defense, ANSI/MIL-STD 1815 A, Juanary 1983.

[AdaEnv 84] **Proc. ACM AdaTEC** "Future Ada Environment Workshop", Santa Barbara, California, 17-20 Sept., 1984, *ACM Ada Letters*, Vol. IV, n. 5, 1985.

Asirelli 84] **Asirelli, P., Martelli, M.**, Integrity Constraints, Redundancy and Consistency in Logic Data Bases. CNUCE Int. Rep. C84-24.-1984.

[Asirelli 85a] **Asirelli, P., De Santis, M., Martelli, M.,** Integrity Constraints in Logic Data Bases, to appear in *Journal of Logic Programming*.

[Asirelli 85b] **Asirelli, P., et al.**, The knowledge Base Approach in the Epsilon Project, *ESPRIT Tec. Week,* Brussels, 23-25 Sept., 1985.

[Barbuti 85] **Barbuti, R. and Martelli, M.**, Programming in a Generally Functional Style to Design Logic Data Bases. Submitted for publication.

[Bowen 82] **Bowen, K.A.and Kowalski, R.A.**, Amalgameting Language and Metalanguage in Logic Programming. In *Logic Programming*, K.L. Clark and S.-A. Tarnlund, Eds. (Academic Press, 1982), 153-172.

[Clark 78] **Clark, K.L.**, Negation as Failure, in *Logic and Data Base*, (Gallaire, H., Minker, J., Nicolas, J. Eds), Plenum Press, New York, pp. 293-322,1978.

[DeRemer 75] **DeRemer, F. and Kron, H.**, Programming in-the-large versus programming-in-the-small, *Proc. Conference Reliable Software*, pp.114-121, 1975.

[Furukawa 84] **Furukawa, K. et al.**, Mandala: A Logic Based Knowledge Programming System, *Proc. Int. FGCS '84* , Tokio, pp. 613-622, 1984.

[Gallaire 84] **Gallaire, H., Minker, J., Nicolas, J.** , Logic and Databases: a deductive approach, *Computing Surveys,* 16, (2), pp. 153-185, 1984.

[Hunke 81] **Hunke, H. ed.**, *Software Engineering Environments*, North Holland, 1981.

[Inverardi 83] **Inverardi, P., Montanari, U., Levi, G., Vallario, G.N.**, A KAPSE Distributed Architecture, *Proc. on Ada Europe- Ada Tec Conference,* Brussels, 16-17 March, 1983. Also *ACM Ada Letters*, Vol. III, n. 2, pp. 55-61.

[Inverardi 85] **Inverardi, P., Mazzanti, F. , Montangero, C.**, The Use of Ada in The Design of Distributed Systems, *Proc. Ada International Conference*, Paris, May 1985.

[Kowalski 74] **Kowalski, R. A.**, Predicate Logic as a Programming Language, *Proc. IFIP 74*, pp. 569-574.

[Kowalski 79] **Kowalski, R.A.**, *Logic for Problem Solving*, North Holland, Artificial Intelligence Series, N.J. Nilsson (Ed.), 1979.

[Liskov 83] **Liskov, B. and Scheifer, R.**, Guardians and actions: Linguistic support for robust distributed programs, *ACM TOPLAS*, vol.5, no.3, pp.381-404, July 1983.

[Lloyd 84] **Lloyd, J.**, *Foundations of Logic Programming*, Springer Verlag, 1984.

[McDermott 81] **Mc Dermott, J.**, R1 The Formative Years, *AI MAGAZINE*, Summer 1981.

[Mitchell 79] **Mitchell, J.G., Maybury, W. and Sweet R.**, Mesa language manual version 5.0, Xerox Palo Alto Research center, Rep. CSL-79-3, April 1979.

[Nicolas 82] **Nicolas, J.**, Logic for Improving Integrity Checking in Relational Data Bases, *Acta Informatica* 18, 227-253, 1982.

[Robinson 65] **Robinson , J. A.**, A machine-oriented Logic Based on the Resolution Principle, *JACM,* 12 (1), pp. 23-41, 1965.

[Shapiro 83] **Shapiro, E. Y. and Takeuchi, A.**, Object-Oriented Programming in Concurrent Prolog, *New Generation Computing*, 1,1, 25-48, 1983.

[Shepherdson 84] **Shepherdson, J. C.**, Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption. *J. of Logic Progr.*, 1, 1,pp. 51-79, 1984.

APPENDIX

Ada Configuration Environment definition

Assertions (due to the interaction with the editor)

subp_decl(a,b.nil)
gen_decl(c,d.nil)
pack_decl(b,nil)
pack_decl(d,nil)
subp_body(a,c.nil)
pack_body(b,nil)
...
sub_unit (fd,h.nil,d.nil)

Assertions (due to the interaction with the compiler)

compiled_sec(a)
compiled_lib(a)

Rules

lib (X,Y) ← subp_decl(X,Y)

lib (X,Y) ← gen_decl(X,Y)

lib (X,Y) ← pack_decl(X,Y)

main (X,Y) ← subp_body(X,Y)

lib_unit (X,Y) ← lib (X,Y)

lib_unit(X,Y) ← main (X,Y)

sec_unit(X,Y,nil) ← subp_body(X,Y)

sec_unit(X,Y,nil) ← pack_body(X,Y)

sec_unit(X,Y,Z) ← subunit(X,Y,Z)

```
parent(X,Y)              ←   main(X,Z), sec(Y,W,X.nil)

parent(X,Y)              ←   sec_unit(X,Z), sec_unit(Y,W,X.W1),
```

Note: *parent* : The definition is splitted in two in order to capture: i) the case in which a main is the root of a parent's chain and ii) the case in which the root is a secondary_unit.

Program Library definition

```
prog_lib(A,Y)            ← compiled_list(Y)
```

Rules used for compilation

```
compiled_list(nil)

compiled_list(X.Y)       ← compiled(X), compiled_list(Y)


compiled(X)              ← compiled_lib(X)

compiled(X)              ← compiled_sec(X)


compilable_lib(X)        ←   lib_unit(X,Y), compiled_list(Y)


compilable_sec(X)        ←   sec_unit(X,Y,nil), compiled_lib(X), compiled_list(Y)

compilable_sec(X)        ←   sec_unit(X,Y,Z), parent(W,X), sec_unit(W,W1,W2),
                             compiled_sec(W), compiled_list(Y)

compilable_sec(X)        ←   sec_unit(X,Y,Z), parent(W,X),
                             main(W,W1),compiled_lib(W),
                             compiled_list(Y).
```

Constraints

lib_unit (X,Y), lib_unit (Z,W) $\rightarrow$ X $\neq$ Z

sec_unit(X,Y,Q), sec_unit(Z,W,V) $\rightarrow$ X $\neq$ Z

sec_unit(X,Y,**nil**), main(Z,W) $\rightarrow$ X $\neq$ Z

sec_unit(X,Y,W), subp_body(X,Y) $\rightarrow$ lib_unit(X,Z), subp_decl(X,Z)

Transactions

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

compile (X,sec) $\leftarrow$ compilable_sec(X) | ***comp_Ada*** (X),**add**(compiled_sec(X))

compile (X,lib) $\leftarrow$ compilable_lib(X) | ***comp_Ada*** (X), **add**(compiled_lib(X))

add 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.

Recompile

recompile (X,sec) $\leftarrow$ compiled_sec(X) | **remove**(compiled_sec(X)), *remove_child_of* (X),*compile* (X,sec).

recompile (X,lib) $\leftarrow$ compiled_lib(X) | **remove**(compiled_lib(X)), *remove_lib* (X),*compile* (X,lib).

remove_child_of (X) $\leftarrow$ parent(X,Y), compiled_sec(Y) | **remove**(compiled_sec(Y)), *remove_child_of* (Y), *remove_child_of* (X)

remove_child_of (X) $\leftarrow$

remove_lib (X) ← compiled_sec(X) | **remove**(compiled_sec(X)),
remove_child_of (X), *remove_connect_lib* (X),
remove_connect_sec (X)

remove_lib (X) ← *remove_connect_lib* (X), *remove_connect_sec* (X)

remove_connect_lib (X) ← compiled_lib(Y), lib_unit(Y,W), member(W,X) |
remove(compiled_lib(Y)), *remove_lib* (Y),
remove_connect_lib (X)

remove_connect_lib (X) ←

remove_connect_sec (X) ← compiled_sec(Y), sec_unit(Y,W,W1), member(X,W) |
remove(compiled_sec(Y)), *remove_child_of* (Y),
remove_connect_sec (X)

remove_connect_sec (X) ←

remove is the remove operation on the object theory (removes an assertion of kind compiled_...(a)).