Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

A Logic Schema for a Kernel Configuration Environment

Patrizia Asirelli and Paola Inverardi

Progetto finalizzato Sistemi informatici e
Calcolo Parallelo, Sottoprogetto 6, Obiettivo AASS

Nota Interna B4-39

Agosto 1989

A Logic Schema for a Kernel Configuration Environment [1]

Patrizia Asirelli and Paola Inverardi

I E I-CNR,
Via S. Maria, 46 - I56100 PISA, Italy
tel. +39-50-500159

Abstract

This paper falls into the area of *database support for configuration management*. It is concerned with the definition of the data model, in particular, the schema (i.e. the static properties of the data model) for a Kernel Configuration Environment (KCE) that has to be general, i.e. not concerned with a particular kind of existing programming environment or with a particular programming language. Since data modelling, in the application area of configuration environments, requires to define the relevant concepts of that activity, the paper has also an impact on the area of *system modelling*.

We consider the KCE as a tool to be integrated in existing programming environments after a specialization of its data model: The schema we have defined is general enough to meet different requirements; it defines the semantics of already known concepts and notions and of new concepts that have been suggested by specification languages.

1. Introduction

Logic has recently been used, by the authors of this paper, to define two different environments to handle configuration activities. One environment was the Unix one, for which the Make facility was defined [Asirelli 87a]. The other one was an Ada-environment for which logic was used to define the Ada configuration facilities [Asirelli 87b]. To make that experience the authors have used as implementation support a prototype logic DBMS, EDBLOG, running on a SUN machine [Asirelli 88a].

The aim of those two working examples was to see whether logic was suitable and how it could be used in programming environments and, furthermore, what advantages logic could bring in this area of computer science.

Once logic proved to bring many advantages, especially in the configuration area, as shown in [Asirelli 88b], notably formalization and rapid prototyping, the authors have concentrated on the definition of a Kernel Configuration Environment that was not influenced by particular, existing, programming environments or disciplines, and abstract enough to cope with the many different activities in software production, e.g. the design, implementation and code generation activities. Furthermore, a KCE has to be abstract, in the sense of providing general relations among objects thus allowing the definition of concepts that can be further specialized when using the KCE to define a specific one.

Quoting from [Tichy 88] "Configuration management is the discipline of controlling the evolution of complex systems; sw configuration management is its specialization for sw systems". Thus, the *Configuration Manager* is the tool responsible for gathering modules together according to strategies that depend on the phase in which configuration has to be performed.

The authors have not found in the literature any explicit definition of data models, underlying existing database-oriented sw configuration environment systems, although in [Tichy 88] a systematization of concepts is given as a glossary for the area.

Quoting from [Brodie 84], *data modelling* concerns the construction of "a representation of the application that captures the *static* and *dynamic* properties needed to support the desired processes", furthermore: "A *data model* is a collection of mathematically well

[1]Work supported by **Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Sottoprogetto 6, Obiettivo AASS.**

defined concepts" and consists of two components: the static and the dynamic properties; the *schema* "consists of a definition of all application object types, including their attributes, relationships and static constraints"; dynamic properties deal with the specification of operations (transactions).

Thus, facing the problem of configuration activities according to the approach we have taken, i.e. data modelling, reflects a database approach to configuration issues.

We have till now concentrated on the definition of a schema for a *kernel configuration environment data base*, (that is: the objects, their relationships and the static constraints), while the dynamic component of the data model is at present being studied, and it is concerned with the definition of strategies to collect modules together, i.e.the definition of the Configuration Manager.

The work we present is funded by the National Research Council, as part of a national project that aims at defining and developing integrated software development environments (AASS).

The schema we propose is very general and the design focussed on the definition of a kernel data structure that maintain the information, relevant to configuration activities, produced during the various phases of the software life cycle, namely during the design phase, the implementation phase and the coding phase.

The goal is, in fact, to have a configuration system based on such a data base that is able to *consistently* support information at the three levels. The level structure of objects in the data model has been provided to this purpose and it is new, as well as the notions of horizontal and vertical relations, that have been first introduced in the area of specification languages [Burstall 80], and do not exist in already defined and/or working configuration environments such as, e.g.[Estublier 86].

In section 2, a logic description of the proposed schema is given; the description is given directly using first order logic but it would be straighforward to give it in a more *conventional* model, e. g. entity-relationship, and then translate it into logic; in section 3, a notion of *correct* development of a software product is induced by the definition of integrity constraints on each level and among different levels. Such notion is related to the structure of the data model and it is supposed to be used in order to guarantee the correct interactions between the data base and any external tool; section 4, concludes the paper. Finally, in the Appendix a summary of the data model is provided.

2. The configuration schema

In this section we present the schema using Horn Logic clauses for facts and rules while integrity constraints are first order clauses $A1,..., Am \rightarrow B1,...,Bn$ with $n,m \geq 0$ whose informal interpretation is that whenever A1 and ... and Am are true then B1 and ... and Bn must also be true.

We assume that sw objects are identified by unique names within the same *project library*.

In the schema, three *types* of software objects are considered: **design, source** and **derived**. The above three *types* of objects represent the vertical layered structure of the model, they denote the objects at the design, implementation and coding levels respectively. The goal is to integrate in the same DB, design and implementation modules, while maintaining all the relevant relations between them. Relations among objects are then considered as belonging to two groups: vertical relations that connect objects on distinct layers and horizontal relations that connect objects on the same layer.

The terminology here used for modules is

generally taken from [Tichy 88]. Thus, design objects may denote modules at the specification level, while source objects may denote modules that are implemented in some specific application language, say Pascal, C etc. We consider *derived modules* as representing both the "derived" and the "manually derived" modules defined in [Tichy 88]. In both cases, the DB is updated by inserting the object together with all the relations that bind it with other objects already existing in the DB, like the source module from which the given object was *derived.*

2.1 The representation of objects

Objects (e.g.modules) are supposed to be introduced as the result of an external interaction (users, tools etc.) and are represented by means of assertions like:

design(obj($name_1$)) ←
source(obj($name_2$)) ←
derived(obj($name_3$)) ←

where "obj" is a function symbol used to build up terms of the DB; "design", "source" and "derived" are unary predicates, provided by the KCE, that are used to *type* the objects in the DB.

Thus, objects in the DB are defined as follows:

object (X) ← design (X)
object (X) ← source (X)
object (X) ← derived (X)

2.2 Relations among objects

A dependency relation among objects can be either vertical or horizontal:

depend_on (X,Y) ← depend_on_ver (X,Y)
depend_on (X,Y) ← depend_on_hor (X,Y)

A vertical relation binds objects on the different levels (layers):

depend_on_ver (X,Y) ← implem_of (X,Y)
depend_on_ver (X,Y) ← compiled_from (X,Y)

Thus, the relations implem_of and compiled_from are used to set up a correspondence between objects at the specification level, at the implementation and at the coding level, respectively.

Horizontal relations bind together objects that: i) are alternatives in the sw development, or ii) depend "structurally" one from each other.

depend_on_hor (X,Y) ← variat_of (X,Y)
depend_on_hor (X,Y) ← depend_on_rel (X,Y)

depend_on_rel (X,Y) ← combine (X,Y)
depend_on_rel (X,Y) ← import (X,Y)

The relation "variat_of" is of the first kind, (i) above, while "combine" and "import" are of the second kind, (ii) above.

The relation variat-of (i.e. variation of) groups together the representation of variants, revisions, versions etc. We have decided to make no distinction among these notions because we believe that they have the same semantics with respect to configuration management; i.e. they connect sw objects that can be considered alternatives with respect to a certain choice. That choice can be made by means of attributes, that will be discussed later on. Thus, we provide for a general notion that could be further specialized by the user, depending on the context he is using, or on the system he is modelling.

The variat_of relation is defined as follow:

variat_of (X,X) ← object(X).

variat_of_tran (X,Y) ← variat_of (X,Y)
variat_of_tran(X,Y)←variat_of (X,Z),
X=\=Z, variat_of_tran (Z,Y)

The transitivity of the variat_of relation is guaranteed by the definition of variat_of_tran, that will be used from now on. In order to define more precisely the semantics of the

variat_of relation an integrity constraint is given:

```
variat_of_tran(X,Y), X=\=Y → attr(X,Z,Vx),
                    attr(Y,Z,Vy), attrname(Z),Vx /==Vy,
                    valueset (Z,Vx),valueset(Z,Vy).
```

The above integrity constraint establishes that an object is the *variation* of an other object only if the same attribute with a different value is defined for each object.

Note that the definition of variat_of has been given using both rules and integrity constraints.

The general purpose of integrity constraints is to establish a minimal semantics while maintaining the definition in the DB as general as possible. Note that, whenever an attribute is defined for two objects, it does not mean that the two objects are variat_of each other, i.e. the right part of the integrity, does not contribute to the definition of variat-of in the generative sense, on the contrary, it expresses a condition that has to be verified for the objects that are in the relation.

The relation depend_on_rel expresses a hierarchical structuring of a sw object in a specific phase of its life cycle.

The relation combine (X,Y) applies when a module Y is part of the decomposition of a another module X; this relation has to be used when an object is defined as the union of other objects; it is defined as a binary relation since a fixed arity cannot be defined a priori. Hierarchical relations imply that a new information in the DB, with respect to a sw object, must exist, e.g. if an object is atomic or structured. This additional information about a sw object is expressed using *attributes*, e.g.

combine (X,Y) → attr(X, atn(type), structured)

The horizontality of this relation is guaranteed by the following integrity constraints:

```
combine (X,Y), design(X)  → design(Y)
combine (X,Y), source(X)  → source(Y)
combine (X,Y), derived(X) → derived(Y)
```

The relation import (X,Y) expresses the dependency between a module Y that imports a module X. The same constraints apply also for the import relation:

```
import (X,Y), design(X)  → design(Y)
import (X,Y), source(X)  → source(Y)
import (X,Y), derived(X) → derived(Y)
```

2.3 Attributes of objects

Attributes are denoted by the unary predicate **attrname**:

```
attrname(atn(language)) ←
attrname(atn(revision)) ←
attrname(atn(type) ←
```

Attribute values are taken from the corresponding *value set* that has to be defined, either in an enumerative way or else by means of facts and rules:

```
valueset(atn(language), pascal) ←
valueset(atn(language), c) ←
valueset(atn(type), atomic) ←
valueset(atn(type), structured) ←
valueset(atn(revision), N) ← integer(N)
integer(0)←
integer(N)← integer(N-1)
```

A ternary predicate is used to connect an attribute with an object:

```
attr((obj(veronica)), atn(language), pascal) ←
```

Integrity constraints are defined for attributes, too:

- valueset (X,Y) → attrname(X)

In this way it is possible to constraint the existence of a value set to the existence of the corresponding attribute; and it is possible to check it independently from the existence of objects on which the attribute X is defined.

The following integrity constraint, instead, holds only in the opposite situation: i.e. it

applies only after the introduction of objects for which a value for the attribute X is defined.

- attr(X,Z,V) → attrname(Z), object(X), valueset(Z,V)

Furthermore the value of a certain attribute for a sw object must be unique:

- attrname(Z), attr(X,Z,V1), attr(X,Z,V2) → V1=V2

The presence in our model of three different levels put some questions on the inheritance of part of the horizontal relations from one level to another, usually lower, level. Such a problem is well known in Artificial Intelligence with the name of "structured inheritance problem", and it will be discussed in the next section.

One last remark on the data model concerns the notion of configuration of a software product. We have, deliberately, not provided any explicit notion of *configurated product* as the result of a configuration activity. Since several approaches can be taken, an explicit representation of a configurated product as an object, either atomic or structured, in the data model is premature. In other words, to provide a notion of *configurated product* we would also have forced the definition of a configuration strategy (dynamic aspect of the model).

3. A notion of correct development

In this section we discuss the consistency of our schema with respect to the software development process. In order to do that we will consider its first order logic representation. Each level is a logic theory, now we have to consistently relate the three theories, one for each level, (design level: Ts, implementation level: Ti, code level: Tc) in order to get the theory for the whole data model.

The three theories are related by means of the vertical relations that permit passing from one theory to another one.

T_S
$\Downarrow$ *implem_of*
T_i
$\Downarrow$ *compiled_from*
T_c

A graphical representation of the vertical and horizontal structure of the schema is depicted below (fig.1).

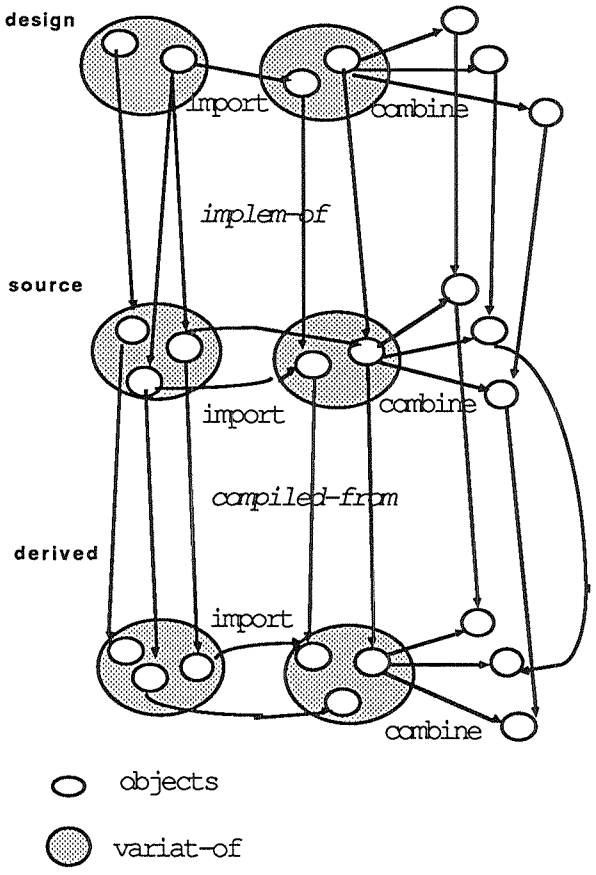

fig. 1

Now we can define our notion of correctness for a software development process by means of the following integrity constraints:

1. variat_of_tran(X,Y), implem_of(X,X'), implem_of(Y,Y'), X'=\=Y' → variat_of_tran(X',Y')

This constraint says (see fig.2) i) that the variat_of relation is inherited at lower levels; ii) that different implementations of the same object must be in the variat_of relation. Note

that, two objects in the variat_of relation may have the same implementation.

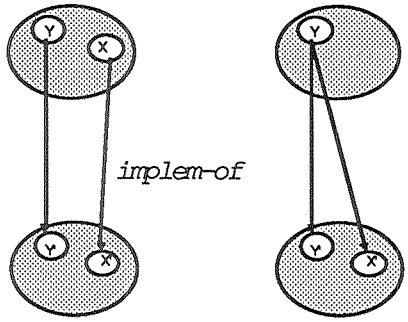

fig.2

2. variat_of_tran(X,Y), implem_of(Z,X), implem_of(Z',Y), Z=\=Z' → variat_of_tran(Z,Z')

This constraint works the opposite direction (lower to higher level, fig.3): it says that if two implementations are in variat_of then their sources must also be in variat_of. The same applies when there is only one implementation for two different sources.

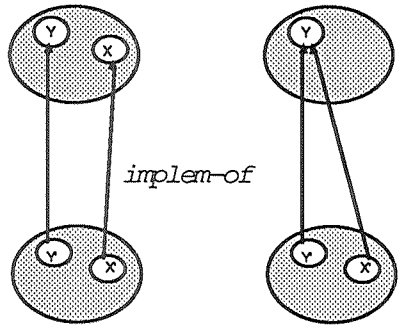

fig.3

The same couple of constraints is defined for the compiled_from relation (see Appendix).

3. import(X,W), variat_of_tran(X,Y), implem_of(Y,Y'), variat_of_tran(W,Z), implem_of(Z,Z') → variat_of_tran(R,Y'), variat_of_tran(S,Z'), import(R,S).

This constraints says that the horizontal relation import has to be inherited at a lower level whenever it holds at the upper level among objects or their variations (fig.4).

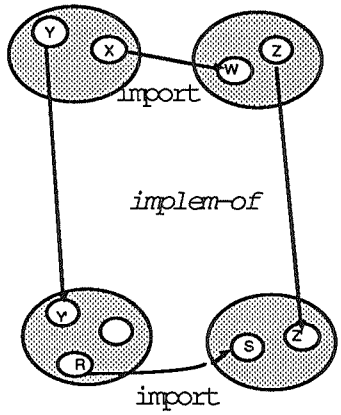

fig.4

A similar constraint, working in the opposite direction, did not seem to reflect the way software is generally developed. In fact, while it is correct to preserve at least the same grain of modularity from a higher level to a lower one; it is usually the case that the granularity is higher at a lower level (top-down development).
The same kind of constraint is defined for the combine relation, too (see Appendix).
Thus we can now express what we meant with *correct development* of a software product: that is when, at any stage of its evolution, the set of integrity constraint (both system and user defined) are satisfied.

4. Conclusions

In this paper we have presented a logic schema for a kernel data base to support configuration management at various stages of the software life cycle. In defining the schema we have been inspired by a number of well known models and systems, even if developed in different contexts. The idea of a consistent vertical and horizontal structure stems from the algebraic specification world, in particular from the Clear language [Goguen 77] and the CAT system [Burstall 80], the same applies also for the structure of the design level, i.e. the relation combine exists in

Clear as well as in Act one [Ehrig 86]. The selection of the relevant relations, wrt configuration management, to be represented at the implementation and at the coding level has greatly benefitted from the clarifying systematization concerning configuration activities presented by Tichy in [Tichy 88].

We concentrated on the definition, of a schema that had to balance between maintaining a certain flexibility of the system while providing the user with non primitive configuration capabilities.

We believe that the use of a tool that forces formalization gives a valid contribution to the definition of the data model; in our case, in fact, it was possible to clarify the different role horizontal relations play with respect to the vertical structure of the model.

The notion of correctness seems to be quite useful since it provides a framework to *validate* the correctness of the *programming in the large* development process in terms of the modification on the data base content it provokes.

Concluding, we can say that our schema seems to be adequate in supporting a kernel configuration environments at least for what concerns the information representation; to become practical it has to be extensively checked when introducing tools, future work, in fact, concerns the introduction of tools and strategies (not only for configuration management purposes) to check the data model effectiveness.

References

[Asirelli 87a] Asirelli, P., Inverardi, P., Enhancing Configuration Facilities in Software Development: A Logic Approach *Proc.1th Europ. Soft. Eng. Conf.*, Strasburg, September 1987, LNCS 289.

[Asirelli 87b] Asirelli, P., Inverardi, P., " A Logic database to support configuration management in Ada" Proc. of the Ada-Europe Int. Conf., Stockholm, 26-28 May 1987, The Ada Companion Series, Camb. Univ. Press.

[Asirelli 88a] Asirelli, P., Inverardi, P., "Edblog: A kernel for configuration environments" *Proc. Int. Work. on Soft. Version and Config. Control*, Grassau, 1988-Jan-28/29. Teubner Ed.

[Asirelli 88b] Asirelli, P., Inverardi, P., Using logic databases in Software Development Environment *Work. on Prog. Lang. Impl. and Logic Prog.: Concepts and Techniques*, 16-18 May 1988, LNCS 348.

[Burstall 80] Burstall, R. M., Goguen, J.A., CAT a system for the structured elaboration of correct programs from structured specification." *Tech. Rep.CSL-118, SRI Int.* 1980.

[Brodie 84] Brodie M. L., On the Development of Data Models, **in** *On Conceptual Modelling*, (Brodie M.L., Mylopoulos J., Schmidt J.W. Eds), Springer-Verlag, 1982, pp. 19-47.

[Goguen 77] Burstall, R. M., Goguen, J.A. , Putting theories together to make specifications, *Proc. Int. Conf. on A.I.*, 1977 .

[Ehrig 86] Ehrig, H., Weber, H., Programming in the Large with Algebraic module specifications, *Proc. Inform. Proces. 86*, North Holland Pub. Co., Amsterdam, 1986.

[Estublier 86] Estublier, J., Adele: a data base of programs. Presentation manual, *Laboratoire de Genie Informatique*, France, June 1986.

[Tichy 88] Tichy, W., Tools for Configuration Management, *Proc. Int. Work. on Soft. Vers. and Config. Control*, Grassau 1988, J.F.H. Winkler Ed., G.B. Teubner-Verlag.

APPENDIX

The KCE data model

object (X) ← design (X)
object (X) ← source (X)
object (X) ← derived (X)

depend_on (X,Y) ← depend_on_ver (X,Y)
depend_on (X,Y) ← depend_on_hor (X,Y)

depend_on_ver (X,Y) ← implem_of (X,Y)
depend_on_ver (X,Y) ← compiled_from (X,Y)

depend_on_hor (X,Y) ← variat_of (X,Y)
depend_on_hor (X,Y) ← depend_on_rel (X,Y)

depend_on_rel (X,Y) ← combine (X,Y)
depend_on_rel (X,Y) ← import (X,Y)

variat_of (X,X) ← object(X).

variat_of_tran (X,Y) ← variat_of (X,Y)
variat_of_tran(X,Y)←variat_of (X,Z),
X=\=Z, variat_of_tran (Z,Y)
attrname(atn(type) ←

valueset(atn(type), atomic) ←
valueset(atn(type), structured) ←

+ application dependent information

design(obj($name_1$)) ←
source(obj($name_2$)) ←
derived(obj($name_3$)) ←
...
attrname(atn(language)) ←
attrname(atn(revision)) ←
attrname(atn(type) ←
...
valueset(atn(language), pascal) ←
valueset(atn(language), c) ←

attr((obj(veronica)), atn(language), pascal) ←

Integrity Constraints

variat_of_tran(X,Y), X=\=Y → attr(X,Z,Vx),
attr(Y,Z,Vy), attrname(Z),Vx /==Vy,
valueset (Z,Vx),valueset(Z,Vy).

combine (X,Y) → attr(X, atn(type), structured)

combine (X,Y), design(X) → design(Y)
combine (X,Y), source(X) → source(Y)
combine (X,Y), derived(X) → derived(Y)

import (X,Y), design(X) → design(Y)
import (X,Y), source(X) → source(Y)
import (X,Y), derived(X) → derived(Y)

valueset (X,Y) → attrname(X)

attr(X,Z,V) → attrname(Z), object(X), valueset(Z,V)

attrname(Z), attr(X,Z,V1), attr(X,Z,V2) → V1=V2

variat_of_tran(X,Y), implem_of(X,X'),
implem_of(Y,Y'), X'=\=Y' → variat_of_tran(X',Y')

variat_of_tran(X,Y), implem_of(Z,X),
implem_of(Z',Y), Z=\=Z' → variat_of_tran(Z,Z')

variat_of_tran(X,Y), compiled_from(X,X'),
compiled_from(Y,Y'), X'=\=Y' →
variat_of_tran(X',Y')

variat_of_tran(X,Y), compiled_from(Z,X),
compiled_from(Z',Y), Z=\=Z' →
variat_of_tran(Z,Z')

import(X,W), variat_of_tran(X,Y),
implem_of(Y,Y'), variat_of_tran(W,Z),
implem_of(Z,Z')
→ variat_of_tran(R,Y'),
variat_of_tran(S,Z'), import(R,S).

import(X,W), variat_of_tran(X,Y),
compiled_from(Y,Y'), variat_of_tran(W,Z),
compiled_from(Z,Z')
→ variat_of_tran(R,Y'),
variat_of_tran(S,Z'), import(R,S).

combine(X,W), variat_of_tran(X,Y),
implem_of(Y,Y'), variat_of_tran(W,Z),
implem_of(Z,Z')
→ variat_of_tran(R,Y'),
variat_of_tran(S,Z'), combine(R,S).

combine(X,W), variat_of_tran(X,Y),
compiled_from(Y,Y'), variat_of_tran(W,Z),
compiled_from(Z,Z')
→ variat_of_tran(R,Y'),
variat_of_tran(S,Z'), combine(R,S).