The Integration of Version Control into Programming Languages

J F H Winkler
Siemens Corporate Laboratories
for Information Technology
D-8000 München 83 Fed.Rep.Germany

Industrial program products are often families of large and modular programs. Modern programming languages support the formulation of such program families only partially. At the time being it is usually not possible to describe different revisions, variants, and versions of single program building blocks and whole programs.
This paper presents a proposal for the formulation of such version information as part of the program text. In a newly introduced CONFIG part of a program building block the programmer can express: (1) to which versions the building block belongs, and (2) which versions of other building blocks it uses. In this new construct a version is defined as a pair (revision, variant), and a variant as a vector of attributes. With these language constructs the "knowledge" about the program versions can be expressed by facts and rules. The representation of this knowledge is adapted to the structure of the program, and the generation of specific program versions can be done automatically.

1 Introduction

The development of programming languages in the last 35 years is characterized by a steady increase of expressiveness. We understand expressiveness not in a purely theoretical sense, but rather in a more practical sense. The first languages before 1960 contained elements for the formulation of mathematical formulae, functions and procedures, and data of certain basic types [Bac 78a; PS 59]. Ten years later Algol 68 and Pascal added the user-defined type, enumeration, record and pointer types; and dynamic data structures [Wi 69a; Wir 71a]. Since then, concepts for modular programming, separate compilation, and concurrency have been added [CCI 85; MMS 79; Ref 83; Wir 80].

Industrial program products are often families of rather large programs. The features incorporated into programming languages during the last ten years support especially the construction of large and modular programs. But, contemporary programming languages do not yet provide constructs for the formulation of program families with revisions and variants, as they are mentioned in [Win 85]. Large and long-lived program products are typically such families. During a long period of usage, such programs usually evolve into a number of revisions [BL 76; Tic 85] (sometimes also called versions or releases [BL 76]). Large program systems are often used for and adapted to a range of functionally different purposes. Operating systems are often adapted to different machine configurations and/or

different user requirements [SW 80], and programs for switching systems are used for different switches in different countries [Win 85; Win 86].

In this paper we propose language constructs to express the information that describes the configurations of program families with revisions and variants. By **revision** we mean different forms of a program that evolve in time during further development and maintenance. By **variants** we mean different forms of a program, that coexist in time and are rather functional alternatives. The distinction between revision and variant is somewhat fuzzy [Loc 83], but it is typically made in practice. The language constructs proposed in this paper describe

a) the revisions and variants a certain program building block belongs to, and

b) the revisions and variants of a building block B that is referenced by another building block A. The revisions and variants of B may depend on those of A.

We use the term "building block" to refer to those entities that are called "compilation unit" in Ada [Ref 83].
The information expressed by the descriptions mentioned above is sufficient for the automatic configuration of arbitrary program versions. In this paper we deal with the problem of program configuration at the building block level. [SW 80] reports about work on configuration at the statement level using program generation techniques.

The paper is organized as follows. Section 2 deals with families of large programs and section 3 with approaches to configuration control. In section 4 we propose a formal model of configuration descriptions. Section 5 deals with the incorporation of configuration descriptions in programming languages. Some examples are given in section 6, and section 7 offers some conclusions.

2 Families of Large Programs

As mentioned in the introduction, large program products are typically families of large programs where the members of such a family are characterized by revisions and variants.

A typical example for such a program family is the EWSD system (electronic digital switching system [Sie 81]). This is a family of CHILL programs for public telephone switches. Its main characteristics with respect to the topic of this paper are:

c1) it is quite large (several Mio Loc and several thousand modules)

c2) it is expected to be in use for several decades. This will lead to a number of revisions.

c3) it is used for functionally different switches as eg local switches and long distance switches. This results in a number of functionally different variants.

c4) it will be installed in a number of different countries worldwide (21 up to now). This results also in a number of functionally different variants.

The process of developing and maintaining such families of large programs has the following three main characteristics:

p1) the programs are **multi-module** programs

p2) the development requires a **multi-library** system as the central database

p3) the projects for such developments are **multi-person** projects.

3 Configuration Control

Programming language facilities that support the realization of program families must take into account the properties and characteristics mentioned in the preceding section. We will generally assume that the programming languages used for the realization of program families allow the formulation of modular programs, like in Ada, CHILL, Mesa, and Modula-2. Since the term "module" is used in some languages for a specific kind of program unit, we will use the term "building block" for the textually self-contained program parts of a modular program.

The set of building blocks of a modular program, or a family of such programs, is a structured set in which certain relations between the elements hold. In the above languages some of these relations are already expressed directly in the source text of the building blocks. Examples of such relations are the relations expressed by the **with**-clauses in Ada, or the export and import clauses in Modula-2. More details are given in [Win 85].

As mentioned, we use the concepts revision and variant to describe the versions of building blocks and whole programs. A revision indication is usually a scalar entity, like a version number or a time stamp. As variant indications we take tuples of attribute values where such a tuple contains a value for some of the attributes that are used to span the space of program variants. In our model a version indication is therefore a pair *(revision indication, variant indication)* i.e. version = (revision,variant) . The formal details of this model are given in the next section.

Contemporary programming languages do not contain facilities to express the version information in a program family. Such facilities will be proposed in section 5.

From the point of programming languages we can distinguish between three different aspects of program configuration:

c1) **Syntactic configuration**: the programming language allows the description of the syntax of the interface of building blocks. This is the situation in programming languages like Ada, CHILL, and Modula-2.

c2) **Semantic configuration**: here the interface semantics is also described by the program text. This is not possible in contemporary programming languages [Win 82], but in the research area there are several approaches to this problem [GHW 85; HKL 84].

c3) **Pragmatic configuration**: the programming language allows additionally the description of the version information as already explained above.

Pragmatic configuration is the topic of this paper. We assume that a programming language allows syntactic configuration as in Ada, CHILL, and Mesa.

4 The Model of the Configuration Language

The configuration language, which will be introduced in section 5, is based on a simple model for the description of versions as pairs of revision and variant.

We assume a finite and nonempty set REV of **revision** indications:

$$REV := \{r_1, r_2, ..., r_n\}$$

REV is often totally ordered, like for time stamps or version numbers . An example is: {1.1, 1.6, 1.8, 2.1, 2.12, 3.0}, where the order is that of the rational numbers.

REV may also be partially ordered as in [Tic 85], where some kind of hierarchically structured revision indications is used, eg: { 1.2, 1.2.1.1, 1.3 }. The ordering is given by: 1.2 < 1.2.1.1 and 1.2 < 1.3 (1.2.1.1 and 1.3 are incomparable). The ordering relation in [Tic 85] expresses a true tree ordering. It is not possible to express the fact that a certain revision R combines different earlier revisions. With a general partial ordering such a recombination (merging [Apo 85: 2-12..2-15]) would also be possible.

A particular building block may belong to several revisions of a program. This is frequently the case if the different versions of a program should be homogeneous with respect to the revision indications [MTW 84]. Therefore, a revision description is defined as a subset of REV. The set of revision descriptions RevD is therefore defined as follows:

$$RevD := Pow(REV)$$

where Pow(M) is the power set of a given set M.

The **variant** of a building block is characterized by the values of some of the attributes. We assume a finite and nonempty set of attributes ATT that are used to characterize the variants of a program family:

$$ATT := \{a_1, a_2, ..., a_m\}$$

Examples for attributes are "speed", "country", "kind", or "type-of-exchange".

For each attribute there is a finite and nonempty set of attribute values:

$$AV_1, AV_2, ..., AV_m$$

Examples for attribute values are "high", "low", "BRD", "USA", or "local".

A variant indication is a partial mapping that maps attributes into the corresponding sets of attribute values. The set of variants or variant indications VAR is therefore:

$$VAR := \{f \in ATT \rightsquigarrow AV \mid \forall a_i \in dom(f) : f(a_i) \in AV_i\}$$

where $\rightsquigarrow$ denotes the set of partial mappings, and $AV = \bigcup AV_i$.

An example for a variant indication is: {(country, BRD), (speed, high)}. A certain building block may belong to different variants and therefore the set of variant descriptions is

$$VarD := Pow(VAR)$$

A **version** is a combination of revision indication and variant indication and the set of versions is therefore:

$$VERS := REV \times VAR$$

An example of a version is: (2.12, {(country, BRD), (speed, high)}).
The set of version descriptions is:

$$VerD := Pow(VERS)$$

The building blocks of a program family reside in one or several libraries. If they reside in more than one library, references to external building blocks must also indicate the library the building block is supposed to reside in [Win 85]. In this paper we assume that the building blocks of a certain program family reside all in one library, ie we do not treat the problems that arise in a multi-library system. The library elements are identified in the library by a name and a version description. Therefore, the set of library elements LE is :

$$LE := N \times VerD$$

where N is the set of names. The names may be simple identifiers, (eg "QuickSort"), structured identifiers (eg "JFHW.QuickSort"), or may include the whole parameter and result type profile [Ref 83: 6.6]. In the rest of the paper we use only simple identifiers .

A library element $e = (n,v)$ determines a set of building block versions BBV(e), where a building block version is a triple (name,revision,variant). For $e = (n,v)$ BBV(e) is defined by:

$$BBV(e) := \{(n, r, a) \mid (r, a) \in v\}$$

A triple (n,r,a) must uniquely identify a library element in a library. This leads to the library consistency condition :

$$LCy(L) := (\forall a, b \in L : a \neq b \Rightarrow BBV(a) \cap BBV(b) = \varnothing)$$

where $L \subseteq LE$ is a library. In terms of data base technology this means that the triple (name, revision, variant) is a primary key [Cod 70a] in L. In the rest of the paper we assume that for all elements of libraries $BBV(e) \neq \varnothing$ holds.

A building block A may refer to (use) another building block B. These references define the use-relation $U \subseteq LE \times LE$. A library $L \subseteq LE$ is called closed if the following library closure condition holds :

$$LCl(L) := U \subseteq L \times L$$

The configuration of a certain program version can be done automatically. This process starts with the building block version that is the so-called main program (eg in the sense of Ada [Ref 83: 10.1]). The rest of the program is selected automatically by the external references. The set of building block versions belonging to the program to be configured is determined by the transitive hull of the external references beginning in the main program.

An external reference is a reference between separate building blocks. An example are the **with**-clauses in Ada [Ref 83]. Such references between building blocks exist also in programming languages in which single entities are exported and imported, as eg in CHILL and Modula-2. If entity E_A, that is contained in building block A, uses entity E_B, that is exported by building block B, then A refers to B. Thus, the uses relation between building blocks is obtained from the corresponding relation between entities by a factorization process.

The configuration process is based on external references in such a way that external references also indicate which version of the external building block is required. In this selection process the version of the referenced building block may depend on the version of the referencing one. Let A be a building block with version (r_s,a_s) (the selecting building block) and E be a building block with version (r_e, a_e), that is referenced by A. The external reference of A to E requires a certain version of E, which may depend on the actual version of A. The following list gives some possible dependencies between (r_s,a_s) and (r_e,a_e).

a)	$(r_e, a_e) = (r_s, a_s)$	E shall have the same version as A
b)	$(r_e, a_e) = (R, a_s)$	definite revision and same variant
c)	$(r_e, a_e) = (r_s, A)$	same revision and definite variant
d)	$(r_e, a_e) = (R, A)$	definite version required
e)	$(r_e, a_e) = (f(r_s), a_s)$	revision depends on r_s and same variant
f)	$(r_e, a_e) = (r_s, f(a_s))$	same revision and variant depends on a_s
g)	$(r_e, a_e) = f((r_s, a_s))$	version is a function of (r_s, a_s)

5 Incorporation into Programming Languages

In order to integrate the configuration descriptions into programming languages, we must provide clauses to define

a) the versions a certain building block belongs to, and
b) the version that an externally referenced building block must have.

We will group this information in a new CONFIG part of a building block. This leads to the following structure:

```
    [<name>]
CONFIG
    <configuration description>
<spec or block>
```

According to the points a) and b) above the <configuration description> consists of two parts:

<configuration description> =
<version definition> [<definition of externals>]

The <version definition> describes the versions which the actual building block belongs to. The <definition of externals> describes other library elements that are used by the block.

From a theoretical point of view the formal configuration descriptions in sect. 4 would suffice. In order to achieve good usability more convenient and efficient descriptions must be added. If e.g. a certain building block A belongs to all but one of the revisions of REV, this can be expressed by listing explicitly all these revisions. If REV contains more than 10 - 20 elements, this enumeration will be tedious and error prone and the reader will not easily grasp the fact that A belongs to all but one of the revisions. From a practical point of view, it is therefore very useful to be able to express that a certain building block does **not** belong to certain revisions. The same holds for the descriptions of variants and versions.

5.1 Definition of the Versions of a Building Block

The <version definition> describes the versions a building block A belongs to. The following grammar defines the syntax of the <version definition>.

<version definition> =
VERS = <version description> [, <version description>]* ;
<version description> = <revision list>: <variant indication>
<revision list> = <list>
<list> = <pos elems> | <neg elems>
<pos elems> = <pos elem> [, <pos elem>]* | ALL
<pos elem> = <value> [..<value>]
<neg elems> = NOT <value> | NOT (<pos elems>)
<variant indication> = <variant description> [,<variant description>]* | ALL
<variant description> = {<attr> = <set> [,<attr> = <set>]* }
<set> = <pos elem> | <neg elems> | {<list>} | ALL

Examples:

VERS = 1: {Speed = {High, Low}};
VERS = 1: {Kind = QuickSort};

The following remarks hold for constructs that contain no <neg elems>. Negation will be treated later.

With respect to the versions a building block B belongs to the <revision list> and the <variant description> are treated differently. B belongs exactly to those revisions mentioned in the <revision list>. If on the other hand a certain attribute a_j is not mentioned in a <variant description> this attribute is treated as a don't care, i.e. B belongs to all values of AV_j. This strategy is necessary for an efficient an convenient building process. During this process the version information is propagated from the building block, that is designated as the main program, to all building blocks belonging to the program to be build. If the information given in a build command mentions a certain attribute a_j that is not directly mentioned in the <version definition> of the main program but in that of any dependent building block D, the information about a_j must be propagated to D.

A <list> defines a set of values, ie the <list> "p_1, ..., p_l", where the p_i are all <pos elem>s of the form <value>, defines the set $\{ p_1, ..., p_l \}$. The keyword ALL represents the set of all values that are possible at the resp. position. If ALL is a <revision list>, it represents the set REV. If ALL is a <variant indication>, it represents the set VAR, and if it is a <list> after <attr>, it represents the set $AV_{<attr>}$. The interval notation <value>..<value> can be used in case the corresponding set is totally ordered. The descriptions defined by the grammar depicted above are slightly more general than those in section 4. A <variant description> denotes a variant description in the following way. Let $v = \{attr_1 = l_1, ..., attr_q = l_q\}$ be a <variant description> and s_i the sets denoted by the <list>s l_i. For i = 1(1)q $s_i \subseteq AV_{attr_i}$ must hold. Then v denotes the variant description VarSet(v):

$$VarSet(v) := \{ f \in VAR \mid dom(f) = \{ attr_1, ..., attr_q \} \wedge (\forall a_i \in dom(f): f(a_i) \in s_i) \}$$

If a <variant indication> vi contains several <variant description>s $v_1, ..., v_p$, the resulting variant description is :

$$VarSet(vi) := \bigcup_{j=1}^{p} VarSet(v_j)$$

A <version description> denotes a version description in an analogous way. Let vd = r : a be a <version description> and RevSet(r) be the set of revisions denoted by r and VarSet(a) be the set of variants denoted by a. Then the set of versions denoted by vd is:

$$VerSet(vd) := RevSet(r) \times VarSet(a)$$

If a <version definition> vd contains several <version description>s $v_1, ..., v_p$, the resulting version description is :

$$\text{VerSet(vd)} := \bigcup_{j=1}^{p} \text{VerSet}(v_j)$$

The incorporation of NOT into the descriptive framework above leaves us with two problems to be solved:

p1) what is the semantics of this construct, and
p2) in case we choose a semantics where inconsistencies are possible, to check a <version definition> for such inconsistencies.

There are at least three different semantics for NOT possible:

NOT-1

NOT is interpreted in a set theoretical sense as complement operator. In this case it is not guaranteed that a specific version is really excluded from the set of versions of a building block. This can be seen in the next example where a <variant indication> is given:

{ Country = {BRD, USA}, Speed = High },
{ Country = USA, Speed = NOT (High) };

If AV_{Speed} = { Low, Medium, High } the resulting variant description is:

{ {(Country, BRD), (Speed, High)},
{(Country, USA), (Speed, High)},
{(Country, USA), (Speed, Low)},
{(Country, USA), (Speed, Medium)} }

because NOT(High) = {Low, Medium}.

This variant description contains that very variant that we intended to exclude: {(Country, USA), (Speed, High)}. We therefore call this interpretation of NOT a **weak negation.**

NOT-2

A second interpretation preserves the idea of complementary sets, which was originally the motivation for the incorporation of NOT, but avoids such inconsistencies as shown above. According to the grammar given above a <version description> has the following form: r:var_1, var_2, ... This is merely a shorthand for r: var_1, r: var_2, ... Therefore we may restrict the discussion to the simple normalized form where a <version description> is of the form

$$r : \{attr_1 = v_1, ..., attr_q = v_q\} \quad \text{for some } q \geq 1.$$

r is either r_s or NOT r_s for some set $r_s \subseteq REV$, and the v_i are either av_{j_s} or NOT av_{j_s} for some set $av_{j_s} \subseteq AV_{attr_j}$. We call an element negative if it begins with NOT and positive otherwise. A <version description> of this simple form, that contains positive elements only, is a positive version, and one containing at least one negative element is called a negative version. The version set VS of a simple <version description> vd is defined as follows:

$$VS(vd) := \text{IF vd is positive} \Rightarrow VerSet(vd) \;\square\; \text{vd is negative} \Rightarrow Compl(vd)\ \text{FI}$$

$$Compl(r : v) := Comp(r) \times Comp(v)$$

$$Comp(r) := \text{IF r is positive} \Rightarrow r \;\square\; \text{r is negative} \Rightarrow REV - r\ \text{FI}$$

$$Comp(\{a_1 = v_1, \ldots, a_q = v_q\}) := \{f \in VAR \mid dom(f) = \{a_1, \ldots, a_q\} \wedge (\forall i \in \{1, \ldots, q\} : \text{IF } v_i \text{ is positive THEN } f(a_i) \in v_i \text{ ELSE } f(a_i) \notin v_i \text{ FI})\}$$

(We assume that both v_i and $AV_i - v_i$ are nonempty.)
Note that Comp is overloaded on revisions and variants.

$$ExclSet(vd) := \text{IF vd is positive} \Rightarrow \varnothing \;\square\; \text{vd is negative} \Rightarrow VS(Pos(vd))\ \text{FI}$$

Here is Pos(vd) = vd', where vd' is obtained from vd by eliminating all NOTs.

With these definitions we can define the consistency of a <version definition>. Let vd be a <version definition> and let $vd' = v_1, \ldots, v_s$ be its normalized form. Then vd is called consistent iff

$$(\forall i \in \{1,\ldots,s\} : v_i \text{ is negative} \Rightarrow \forall j \in \{1,\ldots,s\} : ExclSet(v_i) \cap VS(v_j) = \varnothing))$$

It is easy to see that any <version definition> containing the <variant indication> given in the paragraph NOT-1 would not be consistent.

If vd is consistent the version set determined by vd is given by $\bigcup_{i=1}^{s} VS(v_i)$.

An example for a consistent <version definition> is:

VERS = 1.2: {Country = {BRD, USA}, Speed = High},
{Country = USA, Speed = NOT (Medium)};

NOT-3

A third interpretation of NOT is a strong (strict) interpretation, in which a negative version is always excluded, even in such cases that are inconsistent in the sense of NOT-2. For such a strict NOT two different definitions are possible. Let vd be a <version definition> and $vd' = v_1,...,v_s,v_{s+1},...,v_t$ be its normalized form, where $v_1,...,v_s$ are all positive and $v_{s+1},...,v_t$ are all negative. For the first definition of a strict NOT we obtain :

$$VS(vd) := \bigcup_{i=1}^{s} VerSet(v_i) - \bigcup_{i=s+1}^{t} ExclSet(v_i)$$

and for the second one

$$VS(vd) := \bigcup_{i=1}^{t} VS(v_i) - \bigcup_{i=s+1}^{t} ExclSet(v_i)$$

In the rest of the paper we will use the semantics of NOT-2 for NOT.

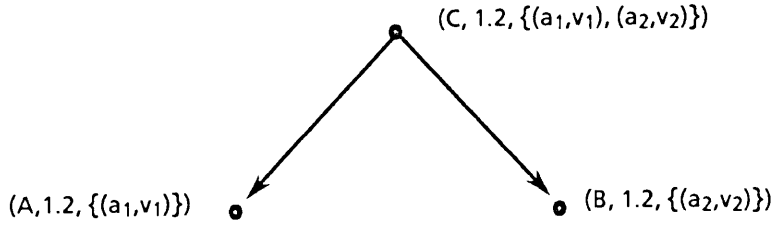
5.2 VERSIONS OF EXTERNAL BUILDING BLOCKS

The selection process starts at the building block that plays the role of the main program. The complete information about the required version of the program to be configured is given in the form of a single version. This information is transitively propagated along the external references in order to select the building block versions that belong to the program to be configured. The information in the variant part of the version description is usually split and distributed over the program. An example of the splitting of the variant part is given in the following Ada example.

```
with A, B;
    -- A is selected by the attribute a1, and
    -- B is selected by the attribute a2 of the
    -- version of C
procedure C is
    -- the version of C is  (1.2, {(a1=v1), (a2=v2)})
```

In a specific situation a_1 may stand for "Device" and a_2 for "Speed".

The example above can be represented by the following program graph:



The variant part of the selecting building block need not be split into disjoint parts; in the general case an external building block may be selected by a subset of the attribute values of the selecting building block.

The splitting and selection strategies mentioned at the end of section 4 lead to the following structure of the <definition of externals>.

```
<definition of externals>  =  <use clause>*
<use clause>               =  USE <id>
                              VERS = ( <revision selection> : <variant selection> |
                                       <version pair> [, <version pair>] *
  -- the list of <version pair>s must define a mapping VERS ⇝ VERS
                                       [, ELSE ⇒ ( SAME | <version> ) ] ) ;
<version pair>             =  <version> ⇒ <version>
<version>                  =  <revision value> : <variant>
<revision selection >      =  SAME | <revision value> |
                              <revision pair> [, <revision pair> ]*
  -- the list of <revision pair>s must define a mapping REV ⇝ REV
                              [, ELSE ⇒ ( SAME | <revision> ) ]
<revision pair>            =  <revision value> ⇒ <revision value>
<variant selection>        =  SAME
                              <variant> |
                              <variant pair> [, <variant pair>]*
  -- the list of <variant pair>s must define a mapping VAR ⇝ VAR
                              [, ELSE ⇒ ( SAME | <variant> ) ]
<variant pair>             =  <variant> ⇒ <variant>
<variant>                  =  {<attr> = <value> [, <attr> = <value>]* [, ELSE = SAME] }
```

Note that (,), [,] are characters of the metalanguage.

The <use clause> describes which version of an external building block E is required by the building block S containing the <use clause> in its <configuration description>. The <id> after USE is the name of the external building block that is being referenced by the <use clause>.

There are two ways to define the version of E that is required by S:

(1) by a pair <revision selection> : <variant selection> where each of the elements may denote a specific revision resp. variant or a mapping in REV resp. VAR;
(2) by a mapping in VERS.

The semantics of the constructs used in the <definition of externals> will be explained together with the building process in the next section.

The USE clause can also be used to select different bodies of packages. The package name used in the CONFIG part of the package specification refers automatically to the corresponding package bodies.

5.3 The building process

The building blocks in the program library are identified by triples (name, revision, variant). The building process starts with a build command of the following form:

BUILD <name> VERS = <revision> : <variant>

The <name> is the name of the main program. <revision> and <variant> determine the exact version of the program to be built. After the exact version of the main program has been selected from the library the <use clause>s in the main program are used to select further building block versions of this program. This process works in a transitive manner until all <use clause>s in the program have been satisfied.

The basic step in the building process is that of the selection of the building block version of a building block E referred to by a <use clause> in building block S. For the following discussion we assume that the actual version of S is (r_a,v_a). This version is transformed in a version (r_s,v_s) by means of the <use clause>, and (r_s,v_s) is then used to select the actual version of E.

We describe the selection process for revisions first. Depending on the different forms of <revision selection> r_s is defined as follows:

(a) <revision selection> = SAME: $r_s := r_a$

(b) <revision selection> = r_u: $r_s := r_u$

(c) <revision selection> is a (partial) mapping $f \in REV \rightsquigarrow REV$:

IF $r_a \in dom(f) \Rightarrow r_s := f(r_a)$
□ $r_a \notin dom(f) \Rightarrow$ *undefined* FI

A library element with name E may belong to a set of revisions R_E. The building process selects that library element with name E for which $r_s \in R_E$ holds. If there is no such library element an error is reported.
In the case of REV being totally ordered we assume a special <revision value> LAST. LAST denotes always the last value of any subset of REV under discussion. In a <use clause>
USE P VERS = LAST: {Country = Norway}; it indicates that for P the latest (newest, most recent) revision of the variant for Norway is to be selected.

For variants the selection step is a little bit more complicated because a variant is a whole tuple of values. Depending on the different forms of <variant selection> v_s is defined as follows:

(a) <variant selection> = SAME: $v_s := v_a$

(b) <variant selection> = $\{a_1 = x_1, a_2 = x_2, ..., a_q = x_q\} =: v_u$:
if v_u does not contain SAME then $v_s := v_u$.
if v_u contains SAME as attribute value the values for these attributes are taken from v_a. If v_a does not contain values for those attributes an error is reported. If no error is reported $v_s := v_u'$, where v_u' is obtained from v_u by replacing SAME.
If ELSE is used in v_u it stands for all attributes mentioned in v_a but not mentioned in v_u.

(c) <variant selection> is a (partial) mapping $f \in VAR \rightsquigarrow VAR$:

IF $v_a \in dom(f) \Rightarrow v_s := f(v_a)$
□ $v_a \notin dom(f) \Rightarrow$ *undefined* FI

Let $A_s = dom(v_s)$. The library elements with name E belong to certain variants v_{ei}. Let $Av_{ei} = dom(v_{ei})$. If there is one of these variants for which $|As \cap Av_{ei}|$ is maximal and v_{ei} is unique in this respect, and if the common attributes have the same values in v_s and v_{ei}, then the library element, to which v_{ei} belongs, is selected with the variant $v_e = v_s \cup v_{ei}$; otherwise an error is reported.

5.3 EXAMPLES

The first example shows a main program MainProgram that uses different variants of a subprogram Sort.

```
        MainProgram
CONFIG VERS =  1 : { Speed  = { High, Low }};
        USE Sort  VERS = SAME: {Speed = High} ⇒ {Kind = QuickSort},
                               {Speed = Low} ⇒ {Kind = BubbleSort};
PROC MainProgram  IS
        ...
        Sort(A, B);
        ...
END  MainProgram;


        Sort
CONFIG VERS  =  1: {Kind = QuickSort};
PROC  Sort(...)  IS
        -- QuickSort algorithm
END Sort ;


        Sort
CONFIG VERS  =  1 : {Kind = BubbleSort};
PROC  Sort(...) IS
        -- BubbleSort algorithm
END  Sort;
```

We can also use the name of a building block as one of the attributes in the variant part of a version. This would be a predefined attribute whereas attributes like "Speed" or "Country" may be freely chosen by the user. The example depicted above then reads like this :

```
     MainProgram
CONFIG  VERS =  1 : { Speed  = { High, Low }};
     USE Sort  VERS  = SAME : { Speed = High } ⇒ { Name = QuickSort },
                              { Speed = Low } ⇒ { Name = BubbleSort};
PROC  MainProgram  IS
     ...
     Sort(A, B);
     ...
END  MainProgram;


   QuickSort
CONFIG  VERS  =  1 : {Speed = High}
PROC  QuickSort(...)  IS
     -- QuickSort algorithm
END QuickSort ;
```

```
        BubbleSort
CONFIG  VERS  =  1 : {Speed = Low}
PROC  BubbleSort(...)  IS
        -- BubbleSort  algorithm
END  BubbleSort;
```

In the next example we see that it is very easy to distinguish between a baseline and a maintenance variant.

```
        CONFIG VERS = 1 : ALL;
                USE P VERS = SAME : SAME;
PROC Main ....

        CONFIG VERS = 1 : {Kind = BaseLine};
PROC P ....

        CONFIG VERS = 1 : {Kind = Maintenance};
PROC P ....
```

With the command

```
BUILD Main VERS = 1 : {Kind = Maintenance}
```

we can build that version of Main that contains the maintenance variants of all building blocks for which such a variant exists.

The version information in a program can be used by other components of the programming environment, too. In order to achieve this some attributes must be predefined as already mentioned in the second example above for the attribute "Name". Another predefined attribute could be "Debug" which could be used to convey information to the compiler. It would then be possible to build a program variant for debugging purposes:

```
BUILD Main VERS = 1 : {Kind = BaseLine, Debug = Yes};
```

6 Comparison with other Work

Program configuration has up to now been carried out by means outside the programming language. The means for program configuration are typically based on facilities provided by operating systems: files and commands for compilation and linking.

The use of files instead of building blocks has the drawback that there is usually no guaranteed correspondence between a file name and the name of the building block contained in that file [Win 85]. Furthermore the operating system based facilities are usually used for languages that do not contain constructs for programming-in-the-large. If those facilities are applied to languages like Ada or CHILL, that contain constructs for programming-in-the-large, this may lead to structural clashes.

The Make facility [Fel 79] is based on the Unix® operating system. A Makefile is a kind of command file and contains two different kinds of elements: (1) elements that describe dependencies between building blocks, and (2) commands that must be executed in order to make (build) the program system. Make is used for C and similar languages, that do not provide constructs for programming-in-the-large. The dependency descriptions allow the description of some aspects of programming-in-the-large, but there remains the problem of the consistency between the source text of the building blocks and the text of the Makefile, e.g. with respect to external references in the source text of the building blocks [Wal 84]. The question of different versions of a program is addressed in Build [EP 84], that is build on top of Make. Different versions of a program are realized in Build by different file directories. There is no facility to characterize the versions by attributes or version numbers. SCCS [Roc 75] and RCS [Tic 85] concentrate on the aspect of storing similar versions of texts efficiently using different kinds of deltas. RCS furthermore uses a state attribute to indicate different variants of a building block.

DSEE (Domain Software Engineering Environment) of Apollo [Apo 85] is one of the most sophisticated configuration tools based on Unix. It distinguishes between a system model and a configuration thread. The system model describes the structure of the program in the large as a block structure in the classical sense, and provides for this a quite elaborated syntax. The configuration thread describes the versions of the building blocks of a corresponding system model. The versions are characterized by version numbers and by version names that are similar to the elements of an enumeration type. These version indications do not provide attributes as they are used in this paper.

The GANDALF system [Hab 85; Not 85] also contains some facilities for version control. There successive versions and parallel versions are distinguished where the former correspond to our revisions and the latter to the variants. The parallel versions are characterized by simple values instead of the tuples used herein. The term "version" is also used with a somewhat different meaning: to distinguish between source, object, documentation etc. belonging to one program unit. Similar as in DSEE a syntax for programming-in-the-large has been developed which is based on boxes and modules [HP 80, 80a] and uses some sort of Module Interconnection Language similar to MIL76 [DK 76].

In the Cedar programming environment [HSZ 85] a system modeller has been built that allows for the description of program configurations [LS 83a]. A system model is a hierarchy of lists of system components, or to be more exact of lists of references to system components. The references refer to the files containing the source text of the building blocks. The file versions are identified by a pair consisting of a name and a time stamp. The models may be parameterized by interfaces as they are defined in Mesa [MMS 79]. There are no further facilities for the description of revisions or variants.

In the Adèle programming environment [Est 85, 86] building blocks are identified by a quadruple (family name, family id, version id, revision number), where the family name corresponds to the usual name, the family id selects a variant in the set of alternatives of the family, the version id identifies a version, and the revision number determines a revision of the version. Version and revision are very similar in this approach: a new version is established after a "major" modification, especially a modification of an interface, and a "small" modification yields a new revision.

The approaches taken in different systems use two different strategies for the organization of the version information: (1) a centralized approach e.g. in Make and DSEE, and (2) a decentralized approach as in Adèle and in our proposal. The main advantage of the decentralized approach is that the representation of the version information is adapted to the program structure. The main advantage of the centralized approach is that the coarse structure of the program can be recognized more easily when inspecting the (centralized) configuration description. But this can also be achieved by a decentralized organization. It is easy to provide a (zooming) function in the programming environment that only displays the CONFIG part of the building blocks.

The main disadvantages of the centralized approach are:
(1) it separates information that belongs together;
(2) it implies the duplication of some information. In the centralized approach the central configuration description must refer to the source modules of the program building blocks. Let B be a program building block that exists in several versions $v_1, v_2,...,v_n$. For reasons of identification the source modules representing the different versions must have different names. If the names are chosen arbitrarily and do not resemble B it is difficult to recognize that they all are versions of B. Thus the names should indicate in some form the identifier B. A similar argument holds for the versions: it would be very useful if the name also contained some hint to the version. The consequence of this is that the name of the source modules should contain some of the version information, i.e. it is quite inevitable that at least part of the version information is repeated in some decentralized representation.

7 Conclusions

The approach in this paper is source oriented. We want to describe different configurations of modular programs using constructs that are integrated into the programming language. We assume that the host language already provides constructs for programming-in-the-large like Ada or CHILL. This distinguishes our approach from most other approaches, that are usually oriented towards languages like C or Pascal. We add to a language with constructs for programming-in-the-large constructs to express versions as pairs of revisions and variants. For the description of variants we provide a very general concept based on tuples of

attributes. The information for program configuration is given where it is used and needed: in the building blocks and the references between building blocks. The constructs described in this paper can be characterized as follows

- the language used to express the version information is a "natural" extension of the programming language;
- a version is a pair (revision,variant);
- the "knowledge" about the program configurations is represented explicitly by facts and rules;
- the representation of this knowledge is adapted to the structure of the program;
- the construction of program versions can be done automatically.

For practical applications it could be useful to define the sets REV, ATT, and AV_i in a central package VERSION.INFO in order to have some control over them. The definition of those sets could be done using appropriate data types like range types and enumeration types. A further enhancement would be the generalization of the information given in the Build command in order to express the fact that certain variants apply to certain building blocks only.

Acknowledgments

The author would like to thank R.Conradi, T. Mehner, and William G.Wood for their constructive comments that led to the improvement of this paper.

8 Literature

Apo 85 Apollo Computer Inc.: DOMAIN Software Engineering Environment (DSEE) Reference Manual. Order No. 003016, Rev. 03, July 85.

Bac 78a Backus, John: The History of Fortran I, II, and III. SIGPLAN Notices 13,8(1978) 165..180.

BL 76 Belady, L.A.; Lehmann, M.M.: A model of large program development. IBM Syst. Journal 15,3 (1976) 225..252.

CCI 85 CCITT: CCITT High Level Language (CHILL), Recommendation Z.200, Geneva 1985.

Cod 70a Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. CACM 13,6(1970) 377..387.

DK 76 DeRemer, F.L.; Kron, H.H.: Programming-in-the-large versus Programming-in-the-small. = [NS 76: 80..89].

EP 84 Erickson, V.B.; Pellegrin, J.F. : Build - A Software Construction Tool. AT & T Bell Laboratories Technical Journal 63, 6(1984) 1049..1059.

Est 85 Estublier, J.: A Configuration Manager: The Adèle Database of Programs. Workshop on Software Engineering Environments for Programming-in-the-Large. Cape Cod, June 85, pp. 140..147.

Est 86 Estublier, Jacky: ADELE - A Data Base of Programs. Presentation Manual. Laboratoire Genie Informatique, 38402 St.Martin d'Heres, Juin 1986.

Fel 79 Feldman, Stuart I. : Make - A Program for Maintaining Computer Programs. Software - Practice and Experience 9 (1979) 255..265.

GHW 85 Guttag, John V.; Horning, James J.; Wing, Jeanette M. : The Larch Family of Specification Languages. IEEE Software 2,5 (1985) 24..36.

Hab 85 Habermann, A. N.: Automatic Deletion of Obsolete Information. The Journal of Systems and Software 5,2(1985) 145..154.

HKL 84 Luckham, David C.; Henke, Friedrich W. von ; Krieg-Brueckner, Bernd; Owe, Olaf: ANNA - A Language for Annotating Ada Programs. Stanford Univ. Comp. Syst. Lab. Tech. Rep 84-261.

HP 80 Habermann, A. Nico; Perry, Dewayne E.: System Composition and Version Control for Ada. Carnegie-Mellon Univ., Dept. of Computer Science, May 1980.

HP 80a Habermann, A. Nico; Perry, Dewayne E.: Well-Formed System Compositions. Dept. of Computer Science, Carnegie-Mellon Univ., CMU-CS-80-117, Pittsburgh, March 1980.

HSZ 85 Swinehart, Daniel C.; Zellweger, Polle T.; Hagmann, Robert B.: The Structure of Cedar. SIGPLAN Notices 20,7(1985) 230..244.

Loc 83 Lockemann, Peter C. : Analysis of Version and Configuration Control in a Software Engineering Environment. = Davis, C.G.; Jajodia, S.; Ng, P.A.; Yeh, R.T. (eds) : Entity-Relationship Approach to Software-Engineering, North Holland, Amsterdam 1983.

LS 83 a Lampson, Butler W.; Schmidt, Eric E.: Organizing Software in a Distributed Environment. SIGPLAN Notices 18,6(1983) 1..13.

MMS 79 Mitchell, James G.; Maybury, William; Sweet, Richard: Mesa Language Manual. Xerox, Palo Alto Research Center, Palo Alto 1979. Version 5.0, CSL-79-3.

MTW 84 Mehner, T.; Tobiasch, R.; Winkler , J.F.H. : A Proposal for an Integrated Programming Environment for CHILL. Third CHILL Conference, Cambridge, September 23-28, 1984, pp. 65..71.

Not 85 Notkin, David: The GANDALF Project. The Journal of Systems and Software 5,2(1985) 91..105.

NS 76 Schneider, H.-J.; Nagl, M. (eds.): Programmiersprachen - 4.Fachtagung der GI, Springer Berlin usw. 1976.

PS 59 Perlis, A.J.; Samelson, K. (eds) : Report on the Algorithmic Language ALGOL. Num. Math. 1 (1959) 41..60.

Ref 83 Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815A. United States Dept. of Defense, Washington, January 1983.

Roc 75 Rochkind, Mark J.: The source code control system. IEEE Trans. on Softw. Eng. 1,4(1975) 364..370.

Sie 81 Siemens AG: EWSD Digital Switching System. telcom report Vol. 4(1981) Special Issue.

SW 80 Winkler, J.F.H.; Stoffel, C. : Methode zur Erzeugung angepaßter und übertragbarer Betriebssysteme.
= Schneider, H.J. (Hrsg.): Portable Software, B.G. Teubner Stuttgart, 1980, pp. 34..47.

Tic 85 Tichy, Walter F.: RCS - A System for Version Control. Software - Pract. & Exp. 15,7(1985) 637..654.

Wal 84 Walden, Kim: Automatic Generation of Make Dependencies. Software - Pract. & Exp. 14,6 (1984) 575..585.

Wi 69a Wijngaarden, A. van: Report on the Algorithmic Language ALGOL 68. Num. Mathematik 14(1969) 79..218.

Win 82 Winkler, J.F.H.: Ada: die neuen Konzepte. Elektron. Rechenanlagen 24,4(1982) 175..186.

Win 85 Winkler, J.F.H.: Language Constructs and Library Support for Families of Large Ada Programs. Workshop on Software Engineering Environments for Programming-in-the-Large. Cape Cod, June 85, 17..28.

Win 86 Winkler, J.F.H.: Die Programmiersprache CHILL. Automatisierungstechnische Praxis 28,5(1986) 252..258, 28,6(1986) 290..294.

Wir 71 Wirth, Niklaus: The Programming Language PASCAL. acta informatica 1(1971) 35..63.

Wir 80 Wirth, Niklaus: MODULA-2. Eidgenössische Technische Hochschule Zürich, Institut für Informatik Bericht 36, March 1980.

CONFIGURATION/VERSION CONTROL

Chair : Per Holager (ELAB-SINTEF, NOR)
Assistant chair : Arne Venstad/Ole Solberg (RUNIT, NOR)

Discussion

Arthur Evans (Tartan Labs, USA):

Mr. Winkler described how to keep the configuration information distributed throughout the source programs, which is an interesting approach. We have been using at Tartan Labs an approach in which we maintain a central database to control our building process. We found that once we got everything sorted out, it was moderately easy to keep things straight. Comments?

Jurgen Winkler (Siemens, FRG):

My approach parallels the one taken in programming languages that provide constructs for PITL, like ADA, CHILL, MESA and MODULA-2. In ADA, the with- and use-clauses are distributed in the program. With this strategy the information is where it belongs. If I inspect a program module, I also want to see its relationships to other modules. If the information is presented in this "object-oriented" manner at the user interface, it does not imply any specific internal structure. Indeed the question of internal structure is not addressed at all.

Jacky Estublier (L.G.I. C.N.R.S., FRA):

The reason why we chose a similar distribution of the configuration information, is that we got fast file access to the object because of good locality of information. The same locality also reduces network traffic and minimizes updating conflicts in a distributed environment.

Andy Rudmik (GTE Comm. Syst., USA):

A question for Ellen Borison on the model for software manufacture. I can see the benefit of coming up with models like this, and I think it is applicable to perhaps a significant part of the life cycle. When I look at the kind of software life cycles in our software development effort, I would characterize the functions as having a high degree of concurrency (needing mutual synchronization), and a lot of non-determinism (not repeatable because of non-recorded user input). I can certainly see that this model might apply to compilation and linking which have fairly straight-forward mapping functions. But a large percentage of the software development activities such as requirements and design involves a high degree of non-deterministic interaction between the various functions. Therefore, I'm suggesting that the model might be extended to incorporate some of these issues.

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

244

Advanced Programming Environments

Proceedings of an International Workshop
Trondheim, Norway, June 16–18, 1986

Edited by
Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

Editors

Reidar Conradi
Tor M. Didriksen
Division of Computer Science, Norwegian Institute of Technology
N-7034 Trondheim-NTH, Norway

Dag H. Wanvik
RUNIT/SINTEF, Norwegian Institute of Technology
N-7034 Trondheim-NTH, Norway

International Workshop on Advanced Programming Environments

Organized by: IFIP Working Group 2.4 on Systems Programming Languages
in cooperation with ACM SIGPLAN/SIGSOFT

Sponsored by: Division of Computer Science (DCS), NTH
Computer Centre at the Univ. of Trondheim (RUNIT)
Kongsberg Våpenfabrikk, Branch Office Trondheim
Royal Norwegian Council for Technical and Scientific Research (NTNF), Oslo
Norsk Hydro, Oslo
Norwegian Teleadministration's Research Lab, Kjeller/Oslo
Integrert Databehandling A/S (IDA), Oslo
International Business Machines, Oslo

CR Subject Classification (1985): D.2.6, K.6.3

ISBN 3-540-17189-4 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-17189-4 Springer-Verlag New York Berlin Heidelberg

Library of Congress Cataloging-in-Publication Data. Advanced programming environments. (Lecture notes in computer science; 244) Papers presented at the International Workshop on Advanced Programming Environments, organized by IFIP's Working Group 2.4 on Systems Programming Languages in cooperation with ACM SIGPLAN/SIGSOFT, sponsored by Division of Computer Science (DCS), NTH and other organizations. 1. Electronic digital computers–Programming–Congresses. I. Conradi, Reidar. II. Didriksen, Tor M. III. Wanvik, Dag H. IV. International Workshop on Advanced Programming Environments (1986: Trondheim, Norway) V. International Federation for Information Processing. Working Group 2.4 on Systems Programming Languages. VII. ACM Special Interest Group in Programming Languages. VIII. ACM Sigsoft. IX. Norges tekniske høgskole. Division of Computer Science. X. Series.
QA76.6.A3327 1986 005 86-31536
ISBN 0-387-17189-4 (U.S.)

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210