Program-Variations-in-the-Small

Jürgen F.H. Winkler, Clemens Stoffel
Siemens AG Munich, Orgasoft Filderstadt

Industrial program products are typically families of rather large and long living programs. Program families evolve due to program-variations-in-the-large (configuration management) and program-variations-in-the-small. Reasons for program-variations-in-the-small are e.g. debug code or variations in data structures as e.g. singly vs. doubly linked list etc. This paper surveys approaches to and techniques for program-variations-in-the-small. It introduces a terminology to describe and classify these approaches. As conclusion it proposes a language oriented approach to configuration management.

Keywords: program families, program-variations-in-the-small, program-variations-in-the-large, program generation, program fragment, reusability, conditional compilation

1 Introduction

Industrial program products are typically families of rather large and long living programs [Win 87]. By a family of programs we understand a set of closely related programs [Par 76; SW 80]. There are different reasons for variations in programs : different functionality, different implementations, program enhancements. We speak of a module family or a subprogram family if we concentrate on variations of a module resp. subprogram.

Similar to the two levels of programming, that are called programming-in-the-small (PITS) and programming-in-the-large (PITL) [DK 76], we deal with program variations also on two corresponding levels: program-variations-in-the-small (PVITS) and program-variations-in-the-large (PVITL). Tools and methods for program-variations-in-the-large are the subject of software configuration management [Apo 85; Roc 75; Tic 88; Win 87]. Program-variations-in-the-large typically deal with the building of program systems out of program building blocks (modules, compilation units). But not all variations can be treated efficiently on module level; e.g. trace or debug code, or variations due to different data structures (e.g. singly vs. doubly linked lists) are typically realized on statement level [Gla 82: 6]. [Boo 87] e.g. presents 104 different variants of queues. Therefore, methods and facilities for program-variations-in-the-small are also necessary. In modern programming languages, that allow for the decomposition of programs into different building blocks, PVITS will typically applied to individual building blocks.

The basic idea of PVITS is to capture different variants, that form a family, in one common piece of code. Ideally, those parts that are common to all members of the family should be represented once and should not be duplicated. Though, in practice a certain degree of redundancy cannot be avoided.

This paper surveys approaches to and techniques for PVITS. It introduces a terminology to describe and classify these approaches. Section 2 deals with the importance of program families in different areas. In section 3 the concepts and terminology for PVITS are introduced. Section 4 presents different technical approaches to PVITS. Section 5 discusses connections between PVITL and PVITS, and section 6 summarizes the techniques presented in sect. 5 and offers some conclusions.

2 The Importance of Program Families

Program families exist in many application areas [Bab 86]. Some examples are:

- Operating systems (e.g. Unix is available for a large variety of machines).
- Database management systems (e.g. ORACLE or INGRES that are available for a sizable number of different computers and operating systems [RS 88]).
- Standard programs for business applications, such as accounting or general ledger systems, are typically in use in different organizations and on different machines.

Typically, a program or program system is determined by two interfaces: the external and the internal interface. The external interface is determined by the user requirements, and the internal interface by the machine the system is implemented on. Usually this machine is an abstract machine realized by a combination of hardware and software. Differences between the different members of a program family are either due to different user requirements or to differences in the underlying abstract machine.

As far as operating systems, programming language processors, or data base management systems are concerned the main reason for building families is the use of different basic machines. One important aspect of these systems is that they shall provide a uniform abstract machine at the user interface while running on different basic machines. On the other hand, the differing user requirements are the main reason for variations in programs for business applications.

Adapting business applications to the variety of user requirements is sometimes called "customizing" [Zim 83]. In program families for business applications the variations are mainly due to different user requirements. At first, these differences lead to a family of conceptual schemas. Dialogue, processing, and reporting functions depend strongly on the conceptual schema in such a way that the variations are mostly variations in the small. The variations in the conceptual schema are often variations in the data structures of the underlying database. If a data structure varies so vary the functions accessing and manipulating this data structure. Since the application program is mostly doing such accesses and manipulations nearly all parts of the program depend on the underlying data structures. As the examples in sect. 5 show variations in access functions for varying data structures are typically on the statement level.

As already mentioned before it is usually not feasible to realize such variants as separate programs. Therefore technical means for the efficient realization of program families are very important.

In some rare cases program systems are planned and developed as program families from the very beginning [Spi 81; SW 80]. In most cases however program families evolve as variants and revisions of an initial program. The main reason for this seems to be that methods and tools for the realization of program families are not yet generally available.

The concept of program families is also an important aspect of reusability. If the common parts of a program family are only developed once and reused throughout the family the cost of development will generally be reduced. Different degrees of reusability are possible. Either the specification or the design and/or some part of the program can be reused. If the specification must be modified the degree of reusability will be low. If only some part of the program must be modified, this indicates a high degree of reusability. A low degree of reusability can be observed if the variations in the program family are due to variations of the user requirements. According to our experience a high degree of reusability is possible if the variations are caused by differences in the basic machine, because the dependences on the basic machine can often be concentrated in a few modules that comprise only 3 - 5% of the whole program code.

When developing and maintaining a program family the program texts and documentation of all family members must be stored and administrated. This can be done in two different ways:

① The documents of each member are stored and managed separately, and development and maintenance are done for each member separately: this results in a a lot of effort if the same modification is to be included in large number of family members. Therefore, this solution is rather infeasible.

② The commonalities in the program family can be exploited in some way. This can be done on the level of program building blocks (PVITL), and on the statement/declaration level (PVITS). If it is done on module level this results in some form of configuration management. If it is done on statement/declaration level we usually obtain some sort of "meta"-program that contains implicitly the different members of the program family. The individual members of the program family are derived mechanically from the "meta"-program. In both cases development and maintenance are done for the whole family. This exploitation of commonalities is a form of reusability.

3 Elements of Program-Variations-in-the-Small

The approaches to PVITS are typically applied to single program building blocks, but do not describe the combination of building blocks into whole programs. In PVITS a family of variants of one program module is usually captured in one single

textual entity. In the following we call such a program a *metaprogram* [FR 82]. [Kru 84] uses the term "multi-version program".

3.1 Language Levels

In a metaprogram we distinguish between two different program levels:

① The level of the code belonging to the elements of the program family captured by the metaprogram. We call this level the *product level* and the language used on this level the *product language* (L_p).

② The level on which we describe how a certain family member is selected from the metaprogram. We call this level the *control level* and the language used on this level the *control language* (L_c).

In some approaches to PVITS the metaprogram acts as a program generator. In this case the programs generated may be formulated in a third language different from L_p. In the general case we therefore distinguish between three different languages:

L_c: the control language.

L_{pi}: the internal product language that is used in the metaprogram.

L_{pe}: the external product language in which the selected family member is expressed.

On the product level we have those parts of the metaprogram that are also parts of the members of the program family. On the control level we describe which parts of the product level belong to a certain family member. On the product level we use the programming language in which the family members are expressed.

The control language is also a programming language. In early realizations of PVITS very scarce languages, that contained only a few very elemental mechanisms, have been used as control language. Experience, e.g. in the PROGES-Project [BWW 76; SW 78, 80], shows that the typical elements of high level languages are useful as elements of a control language. The typical statements of sequential control are used to express the sequence of selection steps, and this selection process is controlled by the values of certain data objects. In a large and complicated control program all the usual data types can be used profitably. Up to now no use of tasking on the control level has been reported.

The activity of developing metaprograms is called *metaprogramming*. In the application of metaprogramming we distinguish between two different activities:

① The construction of the metaprogram.

② The selection of a certain family member out of the metaprogram.

A module family using PVITS is therefore expressed in a mixture of L_c and L_{pi}, and is typically processed in two phases :

① The control program contained in the metaprogram is executed. If this results in another metaprogram (see e.g. [Ler 67]) this program is also executed etc. until the result is a product program P_p.

② The translation of P_p yields an object module form of the family member P_p, that may be linked or executed.

In the following we define four criteria that will be used to characterize and compare different approaches to PVITS. These criteria are:

① kind of selection control;
② relation between L_c, L_{pi}, and L_{pe};
③ legality of the family members;
④ syntactic correctness of the family members.

3.2 Kind of selection control

The selection process determined by the control part of the metaprogram is controlled by the values of certain data objects D. These data objects model the attributes that characterize the different family members and are usually declared on control level. In order to select a specific family member these data objects must be associated with a corresponding tuple of values V. The association with the values of V can be done in two ways:

① Modification of the metaprogram by altering e.g. initializations or assignment statements in the control part. We speak of *internal control* if this is the only way to change the association between D and V. In this case the metaprogram has to be modified each time a different member is to be selected.
② The values of V are provided from outside to the metaprogram. A typical mechanism are e.g. input/output statements in the control language. Another possibility is the Build command in [Win 87]. In this case different family members can be selected without modifying the metaprogram. We speak of *external control* if V can be provided from outside.

If a method allows both for internal and external control it will be classified as "external control" since internal control is always possible by modifying the metaprogram.

3.3 Relation between L_c, L_{pi}, and L_{pe}

We distinguish between

① Fixed combination of L_c, L_{pi}, and L_{pe}, where either $L_{pi}=L_{pe}$ or $L_{pi}\neq L_{pe}$.
② L_c and L_{pi}/L_{pe} are independent of each other, where either $L_{pi}=L_{pe}$ or $L_{pi}\neq L_{pe}$.

3.4 Legality of the family members

As usual, we call a program legal iff it is correct wrt syntax and static semantics (context conditions) of the corresponding programming language. We call a metaprogram P_m *p-legal* iff the legality of P_m implies the legality of all family members (wrt L_{pe}) that can be selected from P_m. We call a metaprogram *m-legal* iff it is not p-legal.

3.5 Syntactic correctness of the family members

We call a metaprogram P_m *ps-correct* iff the legality of P_m implies the syntactic correctness of all family members (wrt L_{pe}) that can be selected from P_m. We call a metaprogram *ms-correct* iff it is not ps-correct.

4 Approaches to Program-Variations-in-the-Small

The concept of PVITS has been realized by a number of different approaches. They differ in the kind of control language, in the relation between product language and control language, and sometimes also in the product language. In the following we mention approaches applied to assembly languages only briefly and give a more detailed account of the approaches used with high level languages.

4.1 Conditional Assembly

In conditional assembly [Ba 70; Ken 69] the control language consists of conditional jumps and some very simple instructions to manipulate scalar values (typically integer values). The basic idea is that the assembly of the product language and the execution of the control language are intertwined. The metaprogram is conceived as a "master program" that contains all the parts of all possible family members and additional control instructions that direct the scan of the assembler through this "master program".

The selection of variants is controlled by jump instructions (unconditional and conditional). These jump instructions are interpreted during the assembly of the product program. If a condition of a jump is true (always for an unconditional) the assembly process is continued at the goal of the jump, and the instructions after the jump are ignored. The conditions are Boolean expressions. It is usually possible to declare variables of scalar type on the control level and to manipulate them by assignment statements. Loops can be realized by backward jumps. Usually, only internal selection is possible.

A special form of conditional assembly has been realized in GENE-300 [Sie 72]. The metaprogram is a sequence of fragments of the product programs. Each fragment is preceded by a condition (formula of propositional logic) over a certain set of attributes. The program variant selected consists of those fragments whose condition is true for a given set of attribute values. A similar technique is reported in [BKS 88; Kru 84] where it is also applied to high level languages and files containing documentation, and is supported by a special editor.

The metaprograms using conditional assembly are usually only m-legal because there is no guarantee that in a family member the typical context conditions of the L_{pe} hold. The metaprograms are ps-correct because all instructions that may be part of any family member must occur somewhere in the metaprogram as a (continuous) substring and can therefore be checked for syntactic correctness.

4.2 Macroprocessors

With macros different variants of a program P may be obtained by processing P using different libraries of macro definitions L_1,..., L_n. The pair <P, {L_1,..., L_n}> corresponds to the metaprogram in the sense of sect.3. A certain program variant P_i is selected by selecting the corresponding library L_i from the set of libraries.

Macros have been combined with conditional assembly [Fre 66; Ken 69; Sie 84]. This adds a procedure mechanism to the control language of conditional assembly. The sequence of instructions generated by different macro calls can depend on the macro parameters and therefore vary arbitrarily.

Macro processors can be fixed to a specific product language [Ken 69], or can be independent of the product language [Bro 67; Hof 73a; Sie 73; Str 65; Wai 70]. Often the product language in systems using macro processors is assembly language; but macro techniques have also been applied to high level languages [BO 83; Bro 80; Com 79; Lea 66; Ler 67; McI 60].

The metaprograms are usually only ms-correct because macros allow the generation of instructions or program fragments parts of which are the values of macro parameters. The values of those parameters are often strings whose value is not checked during execution of the metaprogram.

4.3 Conditional compilation

The counterpart of conditional assembly for high level languages is conditional compilation that has been realized in a number of compilers. The solution typically consists of a control variable e.g. "omit" whose value may be set to TRUE or FALSE. If the value is TRUE at a certain point in the source program the following source text is skipped by the compiler until a control statement is encountered that switches the value to FALSE [Bur 74].

The C programming language [Ker 78] contains some language elements, the compiler control lines, that act as a control language. The construct for conditional compilation is an if-statement:

```
#if constant-expression
<some language constructs>
#else
<some language constructs>
#endif
```

If the checked condition is true then any lines between #else and #endif are ignored. If the checked condition is false any line between the test and an #else or, lacking an #else, the #endif, are ignored.

The control language for C contains furthermore a macro facility where the macro body consists of a *token-string*.

C allows for external control since the values of objects that occur in control statements can be supplied in the compile command. L_c consists of the preprocessor

lines and the macro calls in the C program. $L_{pi} = L_{pe} = C$. The metaprograms are m-legal because the macro bodies are not checked by the preprocessor for their legality wrt C. For the same reason the metaprograms are only ms-correct.

Another approach to conditional compilation is suggested in Ada® [Ref 83: 10.6]. If certain statements will never be executed the corresponding object code can be omitted, and similarly for declarations of entities. This technique has already been applied in optimizing compilers for other high level languages. In Ada only internal control is possible. L_c is a subset of Ada, and $L_{pi} = L_{pe} =$ Ada. The metaprograms are p-legal and therefore also ps-correct.

A simple facility for conditional compilation is also contained in MAP [Com 79].

Example

As a common example for some of the approaches we take a function that removes the first element with a certain property from a linked list and returns a pointer to this element as function result. The variations are in two dimensions:

① The list may be a singly or doubly linked list.

② The condition that determines the element to be deleted may vary.

The following formulation is in C using the preprocessor facilities. The condition is defined as a parameterless macro named "Condition".

```
struct ListType *Remove (Head)
       struct ListType *Head;
{      struct ListType *P1;
#if ListKind = = Single
       struct ListType *P2;
#endif
       P1 = Head->Succ;
#if ListKind = = Single
       P2 = Head;
#endif
while (P1 ! = Head)
    { if (P1->Info Condition) {
#if ListKind = = Double
                P1->Pred->Succ = P1->Succ;
                P1->Succ->Pred = P1 -> Pred;
                P1->Pred = 0;
#else
                P2->Succ = P1->Succ;
#endif
           P1->Succ = 0;
           return (P1); }
        else {
#if ListKind = = Single
                 P2 = P1;
#endif
           P1 = P1->Succ; }
    }
    return (0);
}
```

A specific variant can be selected by compilation in a context where ListKind and Condition are defined accordingly. E.g. :

```
#define ListKind 'S'
#define Single 'S'
#define Double 'D'
#define Condition >100
```

4.4 Preprocessors

Preprocessors vary quite significantly in their approach to PVITS. Simple preprocessors are based on special forms of comments [Bra 87; Sor 86]. In [Bra 87; Sor 86] the special forms of comments are used to generate program variants for different dialects of Pascal. A program part pp belonging to dialect X is written as:

{@X} pp {@}

In this form pp will be treated by the compiler as belonging to the pure program text and belongs therefore to the program variant selected. If the variant for another dialect Y is to be selected the representation of pp must be changed into

{@X pp {@}

i.e. pp is now treated as comment and is therefore not a genuine part of the variant selected. Due to the conventions for comments in Pascal (no nested comments) the technique is of limited applicability. This technique can also be used to capture other variations than those due to different language dialects. Similar facilities, also based on special forms of comments, have been included in other compilers as so-called compilable comments [Sie 83].

[Bra 87] presents a preprocessor COMPAS for Pascal dialects with extensions. Extensions of Pascal usually differ in the syntactic form of the language constructs added to standard Pascal [Win 87a]. The approach of [Bra 87] uses a "neutral" form which is used by the programmer to write the program. The preprocessor then translates the constructs from neutral form into the different dialects. This translation mechanism is no general mechanism to express variations since the preprocessor supports only the translation of a fixed set of language constructs. COMPAS includes also the comment facilities of [Sor 86] as one of its mechanisms.

The mechanisms based on special comments usually provide only internal control since the program must be modified. [Sor 86] provides a preprocessor that changes all special comments to a certain dialect X and therefore provides external control. The relation between L_c and L_{pi}/L_{pe} is fixed. The programs are only m-legal because those parts inside special comments are not checked by the compiler. Therefore, the metaprograms are only ms-correct.

A more general preprocessor is defined by the compile time facilities (CTF) of PL/1 [IBM 72], which have also been applied to other languages [BK 77; Bur 73a; Mus 72, 72a]. The control language of the CTF consists of the following elements:

- variables of type FIXED or CHARACTER
- simple expression
- assignment statement
- goto statement
- IF-statement
- DO-statement
- function subprogram
- include statement

These constructs differ from the corresponding constructs of PL/1 by a leading "%".

With these elements conditional compilation and the effects of macros may be obtained. The function subprograms on control level are somewhat limited and not very well suited for the realization of macros. This can be better achieved by include statements. The control variables serve two purposes: (1) they are used in the control language in expressions etc., and (2) they act as parameterless macros because an occurrence of such a variable in constructs of L_{pi} is replaced by its current value, i.e the body of this macro is variable.

Since there are no I/O statements on control level the CTF offer only internal control. The relation between L_c (% constructs) and L_{pi}/L_{pe} (PL/1) is fixed. The metaprograms are m-legal and ms-correct due to the macro aspects of the variables on control level.

4.5 Program Generation

Program generation is in some sense inverse to the techniques of conditional compilation and the CTF. The basic idea behind these approaches is the insertion of control statements into the product program in order to guide the scan of the language processor through the metaprogram. L_c is usually a rather scarce language that contains not all elements typically found useful in high level languages.

In program generation the view is to have a program (the control program) that is executed and that generates a product program as a result of this execution. This product program may be composed incrementally during the execution of the control program. In program generation L_c is usually seen as a full fledged programming language [FR 82:8].

An early approach to program generation is [Ler 67] a macro generator for Algol 60 [Nau 63]. [Ler 67] uses Algol60 as control language and a slightly modified form of Algol60 as L_{pi}. L_{pe} is a subset of Algol60.

[HP 87] presents a language for the description of families of device drivers. This language contains also language constructs for PVITS. The drivers are automatically generated by processing the family description.

PROGES (PROgramm-GEnerier-Sprache) [BWW 76] is a further approach to PVITS. The main difference between most of the approaches mentioned in sect.3 is

that in PROGES the two levels of programming are more clearly separated, and the control language is richer in constructs as in most of the other approaches.

The first realization of the PROGES concept has the following language constructs:

- scalar data types
- array types
- string types
- variable declaration with explicit initialization
- procedures
- the statements for structuring sequential execution
- a simple IO facility
- a generate statement

PROGES can be characterized as follows: it is a small programming language of the Pascal class with a special statement to generate program parts.

The generate statement generates arbitrary strings of text. Additionally the value of a scalar entity, converted into a text string, or the value of a string variable can be automatically inserted into the generated program part, i.e. this entity acts as a parameterless macro with variable body.

The syntax of the generate statement is:

```
generate-statement  : <$ generate-element* $> .
generate-element    : character* | @ name @ | <| statement* |>
```

The semantics is as follows: The generate-statement appends the program parts obtained by the evaluation of the sequence of generate-elements in the given order to the program generated so far. The evaluation of a sequence of characters yields the same sequence of characters. The evaluation of @ name @ yields the value of name represented as a string. The evaluation of a statement sequence yields the results of the generate-statements contained in that sequence, in the textual order.

With the third alternative of generate-element it is possible to have similar program structures on the control level and on the product level. We may have one generate-statement for the generation of a complete procedure. If there are different variants of this procedure the sequence of generate-elements may contain the statements to control these variants. This is shown in the following example [BWW 76: 38], in which the original product language PEARL [DIN 82] has been replaced by Ada :

```
<$  FUNCTION Remove (Head: in ListType) Return ListType IS
      P1: ListType : = Head.Succ;
      <|  IF ListKind = "Single"
          THEN <$  P2: ListType : = Head;  $>
          FI; |>
    BEGIN
      LOOP  IF P1 = Head
            THEN Return Null;
            END IF;
            IF P1.Info @Condition@
            THEN <| IF ListKind = "Double"
                    THEN <$  P1.Pred.Succ : = P1.Succ;
                             P1.Succ.Pred : = P1.Pred;
                             P1.Pred : = Null;  $>
                    ELSE <$  P2.Succ : = P1.Succ;  $>
                    FI; |>
                 P1.Succ : = Null;
                 Return P1;
            ELSE  <| IF ListKind = "Single"
                    THEN  <$ P2 : = P1;  $>
                    FI; |>
                 P1 : = P1.Succ;
            END IF;
        END LOOP;
    END Remove;
$>
```

The same variant as in sect. 4.3 can be selected as follows. The preceding generate-statement is embedded in a PROGES program in which ListKind and Condition are declared and have the following values at the point immediately before the generate-statement:

ListKind = "Single"
Condition = ">100"

External control is possible because PROGES contains I/O statements. There is no fixed relation between L_c and L_p apart from $L_{pi} = L_{pe}$. The PROGES programs are only m-legal and ms-correct.

PROGES has been used as the control language to develop a family of operating systems for process control computers [SW 80]. The operating systems of this family were oriented toward the support of the real time programming language PEARL [DIN 82]. Furthermore they were also expressed in PEARL (apart from some very small part that interfaced the hardware directly).

In [SW 80] five members of the family are presented. Their length differed from 1230 to 5200 LOC PEARL. This great variance in length was due to different

functionality of these five operating systems. This family of operating systems contained more than one billion variants.

In the family of operating systems the properties of the different members are conceptually characterized by attributes, as e.g.

PEARL-subset: {Full, IITB, MBP, Basic, Θ}
processor: {S310, IITB-P, PDP11}

The attributes are directly modeled as variables of an appropriate type:

e.g.: EXRA("S310", "IITB-P", "PDP11") Processor := Input;

Usually values of different attributes are not independent of each other:

e.g.: Processor ∈ {S310, IITB-P} ⇔ No-of-Reg = 16.

Such conditions can also be expressed in PROGES in a convenient manner since it contains arithmetic and Boolean expressions and the usual statements to express alternatives.

Program generation as treated in this section is different from the typical application generators that are primarily used in business data processing. These application generators as e.g. Nomad or Ramis [HKN 85] are rather compilers for very high level languages. Typical application areas of such generators are generation of user interfaces, generation of report programs, and generation of file update programs. The capture of different program variants in one piece of code is not a goal of these generators.

4.6 Metaprogramming

Metaprogramming is a proposal by Flon, Cooprider, and Raeder [CF 82; FR 82] to overcome some limitations of Ada generics. The generic units of Ada are a macro facility in a high level language. Different instantiations can be derived from a common template. The possible variations are given by a sort of polymorphism: generic units can be parameterized by values, objects, types, and subprograms. In contrast to the usual macro processors the definition of Ada is such that all instantiations of generic units in a legal Ada program result in legal Ada code. On the other hand, the degree of variability of generic units in Ada is somewhat limited. There is no way to have such variations in the body of a generic unit as are present in the example in sect. 4.3. Another gap exists with record types. Record types are not a separate kind of generic parameters, as e.g. array types are. Therefore the flexibility of the generic mechanism is especially limited wrt record types.

The proposal, called Meta-Ada, has the same goals as PVITS in general, i.e. "to express several similar members of a program family in one unifying description" [FR 82: 20] or to capture "different members of a family in one common piece of code, expressing both the similarities and the differences between them" [FR 82: 1].

In Meta-Ada the selection process is directed by the values of options that are declared in the newly introduced meta part of a program unit. The options are constants of discrete type. They are used to model the attributes the different

variants depend on. The control language allows to express that certain parts of a declarative part or of a statement part depend on given values of certain options. Furthermore a mechanism for type extension allows for the extension of generic formal types. With this mechanism it is e.g. possible to extend a generic formal type of the kind *limited* by link fields. This can be used for the realization of flexible list processing packages.

Since the control language of Meta-Ada does not contain any I/O statements only internal control is possible. L_c consists of the constructs that define and use the options. $L_{pi} = L_{pe}$ = Ada, i.e. there is a fixed relation; but this approach could also be applied to other languages. Metaprograms can be p-legal but this requires an in depth analysis of the effects of all possible choices. If the programs are p-legal they are also ps-correct. If the processor of the control part does not perform enough analysis the programs may only be ms-correct.

The example given in sect.4.3 can be expressed in Meta-Ada as follows:

```
META Option ListKind IS (Single, Double);
FUNCTION Remove (Head: in ListType) Return ListType IS
        P1: ListType : = Head.Succ;
    { IF ListKind = "Single" = >
        P2: ListType : = Head; }
BEGIN
    LOOP  IF P1 = Head
          THEN Return Null;
          END IF;
          IF P1.Info > 100
          THEN { ListKind = "Double" = >
                    P1.Pred.Succ : = P1.Succ;
                    P1.Succ.Pred : = P1.Pred;
                    P1.Pred : = Null; ,
                 ListKind = "Single" = >
                    P2.Succ : = P1.Succ; }
                 P1.Succ : = Null;
                 Return P1;
          ELSE  { ListKind = "Single" = >
                     P2 : = P1; }
                 P1 : = P1.Succ;
          END IF;
    END LOOP;
END Remove;
```

The same variant as in sect.4.3 can be selected by the following declaration:

```
FUNCTION MyRemove IS META Remove (ListKind = > Single);
```

4.7 Program Fragments

A more syntax oriented approach to PVITS is based on program fragments that are used in syntax directed program modularization [KMM 83].

A fragment has the following structure:

<name : nonterminal> sentential form

Name is an identifier used to identify the fragment, *nonterminal* is a nonterminal symbol of the grammar G of the programming language used, and *sentential form* is a sentential form that can be derived in G from *nonterminal*. Fragments can be referenced in other fragments by:

<name : nonterminal>

This is the basic mechanism of syntax oriented program modularization.

PVITS can be supported if different variants of a fragment can coexist. In [HKM 86] the *name* is extended by a simple version indication:

name-part . version-part [. subversion-part]

The control language consists of the form clause (given below in the example), the fragment definitions, and the fragment references.

Program fragments provide external control since the selection is initiated by a command similar to the Build command in [Win 87]. This command contains the values for the version and subversion of the variant to be selected. A realization of the concept of program fragments is based on a fixed relation between L_c and L_{pi}/L_{pe}, but the concept can be applied to different product languages. In [HKM 86] the product language is Simula, and in the following example it is Ada. The metaprograms are p-legal if the metaprocessor does the necessary analysis. In this case the metaprograms are also ps-correct by definition.

Using the concept of program fragments the example of sect.4.3 can be expressed as follows:

```
( FORM
      <Remove: subprogram-body>
      FUNCTION Remove (Head: in ListType) Return ListType IS
         P1: ListType : = Head.Succ;
         <P2-decl: object-declaraty>
      BEGIN
         LOOP  IF P1 = Head
               THEN Return Null;
               END IF;
               IF <Required-property : expression>
               THEN    <Remove-from-list: statement-sequence>
                       Return P1;
               ELSE    <Goto-next-elem: statement-sequence>
               END IF;
         END LOOP;
      END Remove; )
```

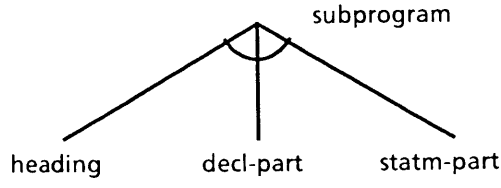
```
( FORM          -- for doubly linked list
      <P2-decl.Double: object-declaraty>
      //
      <Required-property.Double : expression>
            P1.Info < 10
      //
      <Remove-from-list.Double: statement-sequence>
            P1.Pred.Succ : = P1.Succ;
            P1.Succ.Pred : = P1.Pred;
            P1.Pred : = Null;
      //
      <Goto-next-elem.Double: statement-sequence>
            P1 : = P1.Succ; )

( FORM          -- for singly linked list
      <P2-decl.Single: object-declaraty>
            P2: ListType : = Head;
      //
      <Required-property.Single : expression>
            P1.Info > 100
      //
      <Remove-from-list.Single: statement-sequence>
            P2.Succ : = P1.Succ;
            P1.Succ : = Null;
      //
      <Goto-next-elem.Double: statement-sequence>
            P2 : = P1;
            P1 : = P1.Succ; )
```

The same variant as in sect 4.3 can be selected by the directive `Remove [Single]`.

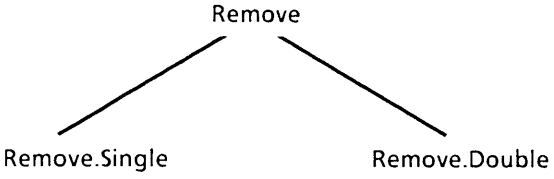
4.8 AND / OR Trees

Programs with variations-in-the-small can be graphically represented by AND/OR trees.

An AND/OR tree is tree with two different kinds of nodes: AND nodes and OR nodes. AND nodes represent the usual structural decomposition used to represent hierarchical program structure or the syntactical structure of programs by syntax trees as in the following example. A syntax tree or abstract syntax tree contains only AND nodes.



OR nodes represent alternatives from which exactly one is chosen. The two forms of the function "Remove", that are due to the different forms of lists, can be represented by the following graph :



AND/OR trees have been used in [SW 78], where they have been called PF-trees (Program Family trees) because they were used to represent program families. They were used as a graphical representation of PROGES programs.

AND/OR trees have also been used in [Har 79] in the form of AND/OR programs. In this approach OR nodes were used for static and for dynamic alternatives. The goal was not primarily the representation of program families but the representation of the development process. Furthermore, AND/OR graphs (acyclic graphs) have been used to represent programs with variations-in-the-large [MTW 84; Tic 82; Win 85].

5 Combination of PVITS and PVITL

A comprehensive and useful system for SVCC should include mechanisms for both PVITL and PVITS. A combination of approaches from both levels should be possible because on both levels versions (revisions, variants) are characterized by some sort of attributes.

The following example combines the configuration language for PVITL of [Win 87] with the metaprogramming proposal of [FR 82]. For this example we assume that the function Remove is contained in the package ListManipulation.

```
ListManipulation
    CONFIG VERS = 1: {ListKind = {Single, Double}};
PACKAGE ListManipulation IS
    TYPE ListType Is ...;
    GENERIC
        With Function Condition(Elem: in ListType) Return Boolean;
    FUNCTION Remove (Head: in ListType) Return ListType;
END ListManipulation ;
```

```
PACKAGE BODY ListManipulation IS
   FUNCTION Remove (Head: in ListType) Return ListType IS
         P1: ListType : = Head.Succ;
      { IF ListKind = "Single" = >
         P2: ListType : = Head; }
   BEGIN
      LOOP  IF P1 = Head
            THEN Return Null;
            END IF;
            IF Condition(P1)
            THEN  { ListKind = "Double" = >
                     P1.Pred.Succ : = P1.Succ;
                     P1.Succ.Pred : = P1.Pred;
                     P1.Pred : = Null; ,
                  ListKind = "Single" = >
                     P2.Succ : = P1.Succ; }
                  P1.Succ : = Null;
                  Return P1;
            ELSE  { ListKind = "Single" = >
                     P2 : = P1; }
                  P1 : = P1.Succ;
            END IF;
      END LOOP;
   END Remove;
END ListManipulation ;

MainProgram
   CONFIG VERS = 1, 2: {ListKind = {Single, Double}, <other attributes>};
      USE ListManipulation DIRECTLY      -- USE = WITH in Ada
         VERS = Same: Same;              -- USE DIRECTLY = USE in Ada
PROCEDURE MainProgram IS
   Head: ListManipulation.ListType ;
   FUNCTION MyCondition (Elem: in ListType) Return Boolean IS
      BEGIN Return (Elem.Info > 100); END MyCondition ;
   FUNCTION MyRemove IS New Remove(MyCondition);
BEGIN
   ...
   ... MyRemove(Head) ...
   ...
END MainProgram ;
```

6 Summary and Outlook

The essential properties of the different methods for and approaches to PVITS are summarized in Table 1. The property "selection control" is independent of the other properties. The legality depends on the L_c/L_p-relation: p-legality usually requires a fixed language realation because the processor for the metaprogram must also know the definition of L_p.

Technique	Selection control	L_c/L_p-relation	Legality	Syntactic correctness
Cond Ass	ext/int	fixed	m-legal	ps-correct
Macro Proc	ext/int	fixed / var	m-legal	ms-correct
Cond Comp	ext/int	fixed / var	m/p-legal	ms/ps-correct
Preproc	ext/int	fixed	m-legal	ms-correct
Progr Gen	external	fixed / var	m-legal	ms-correct
Metaprogr	internal	fixed	p-legal	ps-correct
Fragments	external	fixed	m/p-legal	ms/ps-correct

Table 1. Properties of methods for PVITS

Future programming languages for the construction of industrial program products should contain language elements for both PVITS and PVITL. If we look at the evolution of programming languages we can distinguish different phases with respect to program families. In the first phase (1960-1975) we had programming languages for PITS, that supported PVITS by weak techniques (wrt legality and syntactic correctness). In the second phase (1975-1990) we have now programming languages that support PITS, PVITS (in limited forms), and PITL. But contemporary languages lack elements for PVITL. Therefore, it could be a goal of the next phase of language evolution (1990-2005) to develop programming languages that support both PITS/PVITS and PITL/PVITL.

For PVITS an approach based on parametrization and polymorphism along the lines of generics and program fragments seems to be promising. The advantage of this approach is that the metaprograms are p-legal.

For PVITL flexible and expressive module interconnection languages are necessary that allow to express which versions of which building blocks belong to a certain version of the whole program. The language elements used to express these relationships should be integrated with the rest of the language [Win 87].

The goal of this search for better programming languages is not just a programming or algorithmic language in the classical sense but a language for the formulation of families of large and modular programs.

7 References

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

Ba 70 Barron, D.W.: Assembler und Lader. Hanser, Munich, 1970.

Bab 86 Babich, Wayne A.: Software Configuration Management - Coordination for Team Productivity -. Addison-Wesley Publ. Comp., Reading etc., 1986.

BK 77 Becherer, E.; Kussl, V.: Rationelle Software-Produktion durch mehrstufiges Montieren und Modifizieren von Prozeßrechnerprogrammen. In: Schmidt, Günther (Hrsg.): Fachtagung Prozeßrechner 1977, Springer, Berlin etc. 1977.

BKS 88 Bernstein, R.; Kruskal, V.; Sarnak, N.: Creation and Maintenance of Multiple Versions. In: Winkler, J.F.H.: Proceedings of the International Workshop on Software Version and Configuration Control, B.G.Teubner, Stuttgart, 1988.

BO 83 Brown, P.J.; Ogden, J.A.: The SUPERMAC Macro Processor in Pascal. Software - Practice & Experience 13,4(1983)295..304.

Boo 87 Booch, Grady: Software Components with Ada. The Benjamin/Cummings Publ. Comp., Inc., Menlo Park etc. 1987.

Bra 87 Braunschober, Wilhelm: COMPAS - Compatible Pascal. SIGPLAN Notices 22,3(1987) 78..82.

Bro 67 Brown, P.J.: The ML/I Macro Processor. CACM 10,10(1967)618..623.

Bro 80 Brown, P.J.: SUPERMAC - A Macro Facility that can be Added to Existing Compilers. Software - Practice & Experience 10(1980)431..434.

Bur 73a Burroughs Corporation: System Software Improvements, D Note Documentation, 22 October 1973.

Bur 74 Burroughs Corporation: B 6700/B 7700 ALGOL, Language Reference Manual, 20 May 1974.

BWW 76 Wettstein, H.; Becker-Weimann, K.; Winkler, J.F.H., Wosnitza, H.: Ein modernes, modulares Betriebssystem für Prozeßrechner und seine Generierung. PDV-E71, Kernforschungszentrum Karlsruhe, Juni 1976.

CF 82 Flon, Lawrence; Cooprider, Lee W.: Metaprogramming - Prospects for the Practical Reuse of Software. Univ. of Southern California, Comp. Sci. Dept. TR-112, June 28, 1982, Los Angeles.

Com 79 Comer, Douglas: MAP: A Pascal Macro Preprocessor for Large Program Development. Software - Practice & Experience 9,3(1979)203..209.

DIN 82 DIN Deutsches Institut für Normung e.V.: DIN 66 253 Teil 2. Programmiersprache PEARL, Berlin, Köln Oktober 1982.

DK 76 DeRemer, F. L.; Kron, H. H.: Programming-in-the-large versus Programming-in-the-small. In: Schneider, H.-J.; Nagl, M. (Hrsg.): Programmiersprachen - 4.Fachtagung der GI, Springer, Berlin usw., 1976.

FR 82 Flon, Lawrence; Raeder, Georg: Metaprogramming - Language and Examples. Univ. of Southern California, Comp. Sci. Dept. TR-113, August 4, 1982, Los Angeles.

Fre 66 Freeman, D.N.: Macro language design for SYSTEM/360. IBM Systems. J. 5,2(1966) 62..77.

Gla 82 Glass, Robert L.: Recommended: A Minimum Standard Toolset. SEN 7,4(1982)3..13.

Har 79 Harel, David: And/Or Programs: A New Approach To Structured Programming. In: IEEE (ed.) Specification of Reliable Software. IEEE, New York 1979, pp. 80..90.

HKM 86 Kristensen, Bent Bruun; Madsen, Ole Lehrmann; Møller-Pedersen, Birger; Hauksson, Brynjulv; Nygaard, Kristen: Version control based on syntactic forms. Extended Abstract, Ålborg Univ. Center, Århus Univ., Norwegian Comp. Center, Univ. of Oslo, April 1986.

HKN 85 Horowitz, Ellis; Kemper, Alfons; Narasimhan, Balai: A Survey of Application Generators. IEEE Software 2,1(1985)40..54.

Hof 73a Hofmann, Fridolin: Makroübersetzer für Prozeßrechner 320. Siemens Zeitschrift 47, 1 (1973) 29..33.

HP 87 Herter, Thomas; Pleßmann, Klaus W.: Eine Sprache zur Generierung von Gerätetreibern. Angewandte Informatik 29,8/9(1987)369..378.

IBM 72 International Business Machines Corp.: IBM System/360 Operating System. PL/1(F) Language Reference Manual. No. GC 28-8201-4, 5. ed. Dec 1972.

Ken 69 Kent, W.: Assembler Language Macro Programming: A Tutorial oriented toward the IBM 360. Computing Surveys 1,4(1969)183..196.

KMM 83 Kristensen, Bent Bruun; Madsen, Ole Lehrmann; Møller-Pedersen, Birger; Nygaard, Kristen: Syntax Directed Program Modularization. In: Degano, P.; Sandevall, E. (eds.): Integrated Interactive Computing Systems, North Holland, Amsterdam etc. 1983, pp. 207..219.

KR 78 Kernighan, Brian W.; Ritchie, Dennis M.: The C Programming Language. Prentice Hall, Inc., Englewood Cliffs, 1978.

Kru 84 Kruskal, Vincent: Managing Multi-version Programs with an Editor. IBM J.Res. and Develop. 28,1 (1984)74..81.

Lan 80 Lansky, Amy L.: PASMAC - A Macroprocessor for PASCAL. Stanford University, Computer Systems Laboratory, Techn. Note 174, April 1980.

Lea 66 Leavenworth, B.M.: Syntax Macros and Extended Translation. CACM 9,11(1966) 790..793.

Ler 67 Leroy, H.: A macro-generator for ALGOL. SJCC 1967, 663..669.

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

Mus 72 Mußtopf, G.: Das Programmiersystem POLYP. Angewandte Informatik 14,10(1972) 441..448.

Mus 72a Mußtopf, G.: Sprachgesteuerte Modifikation von Quellprogrammen. In: GI (Hrsg.) Zweite Fachtagung über Programmiersprachen. Saarbrücken 7-9 März 1972.

Nau 63 Naur, Peter (ed.): Revised Report on the Algorithmic Language ALGOL 60. CACM 6, 1(1963)1..17.

Par 76 Parnas, D.L.: On the design and development of program families. IEEE-SE 2,1(1976)1..9.

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.

RS 88 Rohrbach, Roger; Seiwald, Christopher: Galileo: A Software Maintenance Environment. In: Winkler, J.F.H.: Proceedings of the International Workshop on Software Version and Configuration Control, B.G.Teubner, Stuttgart, 1988.

Sie 72 Siemens AG: GENE-300. Programmbeschreibung, Februar 1972.

Sie 73 Siemens AG: Der Prozeßrechner 330. Handbuch Juli 1973.

Sie 83 Siemens AG: FOR1 FORTRAN-Compiler. Order-No. U577-J-Z55-2, D15/5526-04. Munich 1983.

Sie 84 Siemens AG: Assembler BS 2000. Reference Manual. Order No. U62-J-Z55-4-7600, Munich 1984.

Sor 86 Sorens, Michael J.: A Technique for Automatically Porting Dialects of Pascal to Each Other. SIGPLAN Notices 21,1(1986)58..63.

Spi 81 Spitta, T.: Mirekon-ein portables Anwendungssystem. Konzept und Erfahrungsbericht. Angewandte Informatik, 23,10(1981) 446..453.

Str 65 Strachey, C.: A general purpose macrogenerator. Computer Journal 8(1965)225..241.

SW 78 Winkler, J.F.H.; Stoffel, C.: Methode zur Betriebssystemgenerierung und -modularisierung für Prozeßrechensysteme. Kernforschungszentrum Karlsruhe, KfK-PDV153, Februar 1978.

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

Tic 82 Tichy, Walter F.: A Data Model for Programming Support Environments and its Application. In: Schneider, H.-J.; Wasserman, A.I. (eds.): Automated Tools for information systems design. North Holland, Amsterdam etc., 1982, pp. 31..48.

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

Tic 88 Tichy, Walter F.: Tools for Software Configuration Management. In: Winkler, J.F.H.: Proceedings of the International Workshop on Software Version and Configuration Control, B.G.Teubner, Stuttgart, 1988.

Wai 70 Waite, W.M.: The Mobile Programming System: STAGE 2. CACM 13,7(1970)415..421.

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

Win 87 Winkler, J.F.H.: Version Control in Families of Large Programs. 9th International Conference on Software Engineering, Monterey, March 30-April,2 1987, pp. 150-161.

Win 87a Winkler, J.F.H.: Some Improvements of ISO-Pascal. In: Horowitz, Ellis (ed.): Programming Languages - A Grand Tour, Computer Science Press, Rockville, 3rd Ed. 1987, pp. 123..153.

Zim 83 Zimmermann, G.: Customizing von Anwendersoftware. Angewandte Informatik 25,3 (1983) 114..119.

® Ada is a registered trademark of the United States Government (Ada Joint Program Office)

Jürgen F.H. Winkler
Siemens AG
Corporate Research and Technology
Otto-Hahn-Ring 6
D-8000 Munich 83
Fed. Rep. of Germany
winkler@ztivax.siemens.com

Clemens Stoffel
Orgasoft
Weidacher Str. 26
D-7024 Filderstadt
Fed. Rep. of Germany

Berichte des German Chapter of the ACM

Im Auftrag des German Chapter
of the ACM herausgegeben durch den Vorstand

Band 30

Die Reihe dient der schnellen und weiten Verbreitung neuer, für die Praxis relevanter Entwicklungen in der Informatik. Hierbei sollen alle Gebiete der Informatik sowie ihre Anwendungen angemessen berücksichtigt werden.

Bevorzugt werden in dieser Reihe die Tagungsberichte der vom German Chapter allein oder gemeinsam mit anderen Gesellschaften veranstalteten Tagungen veröffentlicht. Darüber hinaus sollen wichtige Forschungs- und Übersichtsberichte in dieser Reihe aufgenommen werden.

Aktualität und Qualität sind entscheidend für die Veröffentlichung. Die Herausgeber nehmen Manuskripte in deutscher und englischer Sprache entgegen.

Proceedings of the

International Workshop on Software Version and Configuration Control

Edited by
Jürgen F. H. Winkler, Siemens AG, Munich

B. G. Teubner Stuttgart 1988

CIP-Titelaufnahme der Deutschen Bibliothek

International Workshop on Software Version and Configuration Control (01, 1988 Grassau):
Proceedings of the International Workshop on Software Version and Configuration Control : (Grassau, FRG, 27/29 Jan. 88) / ed. by Jürgen F. H. Winkler. (Sponsored by German Chapter of the ACM ; GI-Ges. f. Informatik e.V. Referees Dennis Allard ...)
Stuttgart : Teubner, 1988
(Berichte des German Chapter of the ACM ; Bd. 30)
In d. Vorlage auch : SVCC
ISBN 3-519-02671-6
ISSN 0724-9764

NE: Winkler, Jürgen F.H. (Hrsg.); Software version and configuration control; Association for Computing Machinery / German Chapter: Berichte des German ...

Printed in Germany
Gesamtherstellung: Präzis-Druck GmbH, Karlsruhe
Umschlaggestaltung: M. Koch, Reutlingen

International Workshop on

Software Version and Configuration Control

SV CC

Grassau, FRG, 27/29 Jan 88

Sponsored by: German Chapter of the ACM
GI - Gesellschaft für Informatik e.V.

Supported by: Siemens Central Research and Technology

Program Committee:

Gerhard Barth (Univ. of Stuttgart, FRG)
Peter Feiler (SEI, Pittsburgh, USA)
Reinhold Franck (Univ. of Bremen, FRG)
Wolfgang Henhapl (TH Darmstadt, FRG)
David Leblang (Apollo, Chelmsford, USA)
Manfred Nagl (TH Aachen, FRG)
Birger Møller-Pedersen (NR, Oslo, Norway)
Walt Scacchi (USC, Los Angeles, USA)
Walter Tichy (Univ. of Karlsruhe, FRG)
Jürgen Winkler (Siemens, München, FRG) (Chair)

Organizing Committee:

Helmuth Benesch (Siemens)
Rosemarie Weigand (Siemens)
Manuela Wenzl (Siemens)

german acm chapter

Siemens AG
Corporate Research
and Technology