A Generic Bill of Materials based on a Programming Language Notation

Kai A. Olsen, Per Sætre, Anders Thorstenson

Molde College and Research Center Molde,
N-6400 Molde, Norway
kai.olsen@himolde.no, per.satre@himolde.no,˝ anders.thorstenson@himolde.no

Abstract

Generic or general Bill of Materials (BOM) structures make˝ it possible to handle product variants in a convenient way. A programming language for describing such˝ structures is presented. This approach offers a high level of flexibility, both to describe variants˝ and constraints between components in the goes-into relationship. The generic structure is compiled as˝ with any traditional programming language. The user can explode (execute) any part of the BOM to˝ define a specific product variant. User specifications are given dynamically, as the generic BOM is˝ exploded. The system will then generate a conventional BOM for this product variant.

Key words: Bill of Materials (BOM), generic structure, variant˝ handling, BOM generation, programming language, dynamic BOM.

1. Introduction

Modern production is highly customer oriented. Today, many˝ companies try to satisfy demands from their customers through engineer-to-order, produce-to-order˝ or assemble-to-order production systems. The conventional approach to variant handling is to specify˝ each variant as a separate product through an individual Bill of Materials (BOM). This will work with a˝ limited set of variants, but will result in a data explosion if we try to follow a customer-oriented philosophy˝ or if we want to utilize the flexibility of a modern production plant. What is needed is a *generic* BOM description, i.e., *one general BOM* for all product variants.

The generic structure will be used to *generate* a BOM for a particular product variant. That is, for each generic component included in the product we shall give˝ specifications such that one specific variant is selected. By traversing the generic BOM structure, giving˝ specifications where needed, a specific BOM shall emerge. Ideally, all constraints should be specified in˝ the generic structure in order to simplify specification of a product variant. Then this, often˝ frequent, task may be performed by users without expert knowledge of the manufacturing of the product (for˝ example, by sales people).

The principle of generating BOMs have been applied earlier.˝ Wedekind and Müller [1981] introduce a grammar-based method to construct a generic BOM-graph,˝ presented as an extension to a conventional BOM. The BOM-graph consists of logical graphs with˝ “konjunktivknoten” where all children are components of the parent, and “alternativknoten” where˝ one and only one child must be selected as a component. The graph may be useful for presenting legal˝ variants of products and will in this way support the variant description process. A disadvantage with˝ the method is that each variant of a component is represented by a node (e.g., as children of an˝ “alternativknoten”), thus it does not fulfill the requirement of a truly generic approach.

The BOM structure of Schönsleben's [1985]˝ "Variantengenerator" presents all variants of each component. A product variant is then specified by giving˝ values to a set of mutually independent parameters. Of course, in practice there will be dependencies˝ between the parameters. Thus, care has to be taken in the specification process that "illegal"˝ value combinations are avoided.

VanVeen [1992, vanVeen and Wortmann 1992a, 1992b] introduces˝ a generic bill of material concept, an improvement of the "Variantengenerator". This system˝ allows parents to restrict the variability of a child (e.g., the car may control the selection of an engine). On˝ the other hand, the structure will allow for production of a component independently of the way the˝ component is used in higher level products (e.g., we may produce engines independently of their use in˝ cars). The cost of this improvement is a more complex definition of the goes-into relationship, and the˝ inclusion of a conversion function that determines which variant of a child that is to be selected˝ for each variant of a parent. As the syntax used is rather primitive, complex structures will result in˝ elaborate conversion functions. Another problem, which vanVeen discusses in his book [1992], is that it is˝ difficult to express structures where one should choose a variant from one out of several possible˝ sub-components, for example, saying that a car must have a variant of a gas engine *or* a variant of a diesel engine.

Hegge and Wortmann [1991] introduce another method of˝ overcoming the problems with the "Variantengenerator". Here the parent-child connection˝ is achieved through a set of inheritance rules, i.e., conditions that implicitly determine which variant of˝ the child that is to be used for a given variant of the parent. A drawback of this method is that it relies on˝ rather complicated semantics (the inheritance rules, a global name and value scope), and that the˝ parent-child connection is determined implicitly rather than explicitly.

The major weakness of these last systems is that the complete˝ set of parameter values that determines the product variant has to be given in advance, i.e., before the˝ generic BOM is exploded. In opposition to the "Variantengenerator," both systems can, to a certain extent,˝ validate the set of parameter values given. However, apart from this they do not support the˝ specification process.

Bottema and van der Tag [1992] describe a system where this˝ deficiency is partly overcome, as values for product variant features are entered during the˝ configuration process. However, thir system require a post-validation of user choices, which makes it cumbersome to˝ use.

2. A programming language for describing a generic BOM

A bill of material is a *data structure*. The specific (traditional) BOM is made up of *constants*, while the generic BOM includes *variables*. These are all terms that lead towards programming˝ languages. Of course, a programming language is much more than a notation˝ for expressing data structures, but data description has become an important issue of all modern˝ languages. In the following, we shall show that these languages may give important inspiration as to how a˝ generic BOM system may be constructed.

The idea of matching programming languages and BOM˝ descriptions is not new. Blaha et al. [1990] combine elements of databases, object-oriented principles and˝ expert systems for product specification. Chung and Fisher [1992, 1994] construct an object-oriented˝ data model for a BOM by utilizing the sub-class construct to represent goes-into relationships, where˝ subcomponents inherit the features of their parents. While we recognize the power of an object-oriented˝ approach, we cannot see that the composite object hierarchy and inheritance features of such systems are˝ advantageous for modeling product structures. The main problem of this approach is that˝ components cannot be described independently of their utilization. Such independence of description is˝ important in increasing the commonality between products, i.e., we need components that can be used in many˝ products.

We shall take a more modest approach. As we shall see, a˝ generic product description may be constructed just by importing a few constructs from the˝ programming world:

- the procedure concept
- variables
- the input concept
- selection (case) statements

A procedure-oriented approach ensures that the generic BOM is˝ dynamic, in the sense that it is executed as any other program. This execution process has many˝ similarities to the explosion of a conventional BOM. However, while a BOM explosion will result in a full˝ traversal of the structure, the exploding of all components and subcomponents, the traversal of a generic˝ structure will be controlled by user input, selection statements, etc. We shall discuss this variant˝ selection (product specification) in the next section.

Below each of the constructs of our generic BOM specification˝ system will be presented through examples. In these, emphasis is on attribute specifications˝ and the goes-into relationships. A formalized description of the syntax of our programming language is˝ given in the appendix.

The *procedure* concept is applied to describe components of the generic˝ BOM structure. A procedure, here identified by the reserved word **component**, consists of a *head* and a *body*. The head tells us how this component is identified, and presents its *attributes*. The body presents the goes-into relationships. An example of a head declaration is given below, where the seat˝ of a stool is defined as:

```
component §400 is
   name("seat");
   seatColor(red|blue|white);
end component;
```

“Seat” is identified by the number 400. Component˝ identifiers are recognized by a paragraph mark and may consist of letters and/or digits. Seat has the single˝ attribute seatColor. The legal values for seatColor are red, blue and white, or in other words: a variant of seat˝ is identified by the number 400 and one of these three colors. This makes up the component *head* (the body part of the component is not shown). In principle, one should only need the head part of a component˝ description in order to utilize the component in a product.

We may now use seat as a component of another product. This˝ is shown below, where seat is used in a stool:

```
component §200;

body §200 is
   include §400;
   include §500;
end body;
```

Here a short form is used for the head part of the component˝ declaration (the component does not have attributes and the name is omitted). Through the include˝ statement we indicate which (sub)components that go into stool, here – by default – *one* item of 400 (seat) and one item of 500 (base). We˝ have not specified which variant of the generic components that is to˝ be included. This will be done when we expand the generic BOM to specify a product variant of˝ stool. One item of 400 (seat) will then be included. From the declaration of seat it is clear that this˝ component comes in three variants. The system will then ask the *user* to choose between these. In this way we introduce dynamics˝ to our system, as attributes may be specified when needed.

However, stool has the possibility of restricting the˝ possible choices of its sub-components. We then have to use the full version of the include statement, such˝ as demonstrated in the following examples:

```
body §200 is
    include §400 with
      seatColor(red|blue);
    end include;
    include §500;
end body;
```

```
body §200 is
    include §400 with
      seatColor(blue);
    end include;
    include §500;
end body;
```

In the declaration to the left, the legal values for seat,˝ used as a component in stool, are limited to red and blue. During product specification the user will be asked to˝ choose between these two values. In the example to the right, legal values of color have been limited˝ to blue, i.e., to a single variant. In this case, the user will have no influence on the selection of seat˝ variants, as the choice has already been made.

```
body §400 is
   include §410;          -- cushion
   include §420;          -- chipboard
   case seatColor is
      when red:  include §451; -- red seat cover
      when blue: include §452; -- blue seat cover
      when white:include §453; -- white seat cover
   end case;
end body;
```

As seen from the above body part of seat it consists of the˝ components 410 (cushion) and 420 (chipboard). Through the case statement we let the value of˝ the attribute seatColor determine the choice of seat cover. If seatColor is red item 451 is included, etc.˝ Note that comments (prefixed by --) may be inserted freely in the program text.

Alternatively we could treat the seat cover as a generic˝ product:

```
component §450 is
   name("cover");
   coverColor(red|blue|white);
end component;
```

Cover does not have a body part, and therefore no BOM˝ structure. The attribute coverColor is therefore treated as a *specification* of the product variant. Given this declaration the case˝ statement in 400 (seat) could be replaced by:

```
include §450 with
   coverColor(seatColor);
end include;
```

If we have covers of different sizes these could be˝ described by including the attribute size:

```
component §450 is
   coverColor(red|blue|white);
   size(50x60|50x80);
end component;
```

The include statement in 400 (seat) could then give both˝ the color and the size, for example:

```
include §450 with
   coverColor(seatColor);
   size(50x60);
end include;
```

Note that values, such as ‘red’, ‘50x60’, etc. ˝ are represented as strings. Thus 50x60 is a legal value. Normally we will specify the set of legal values for an˝ attribute, as shown in the examples above. In some situations, especially where an attribute is treated as a˝ specification, it can be cumbersome to enumerate all possible values. The system therefore allows for a˝ generic value, indicated by the word *any*. For example, the attribute size of 450 could have been specified˝ as `size(any)`, indicating that all values are allowed.

We shall return to additional constructs of our programming˝ language in Section 4.

3. From a generic to a specific BOM

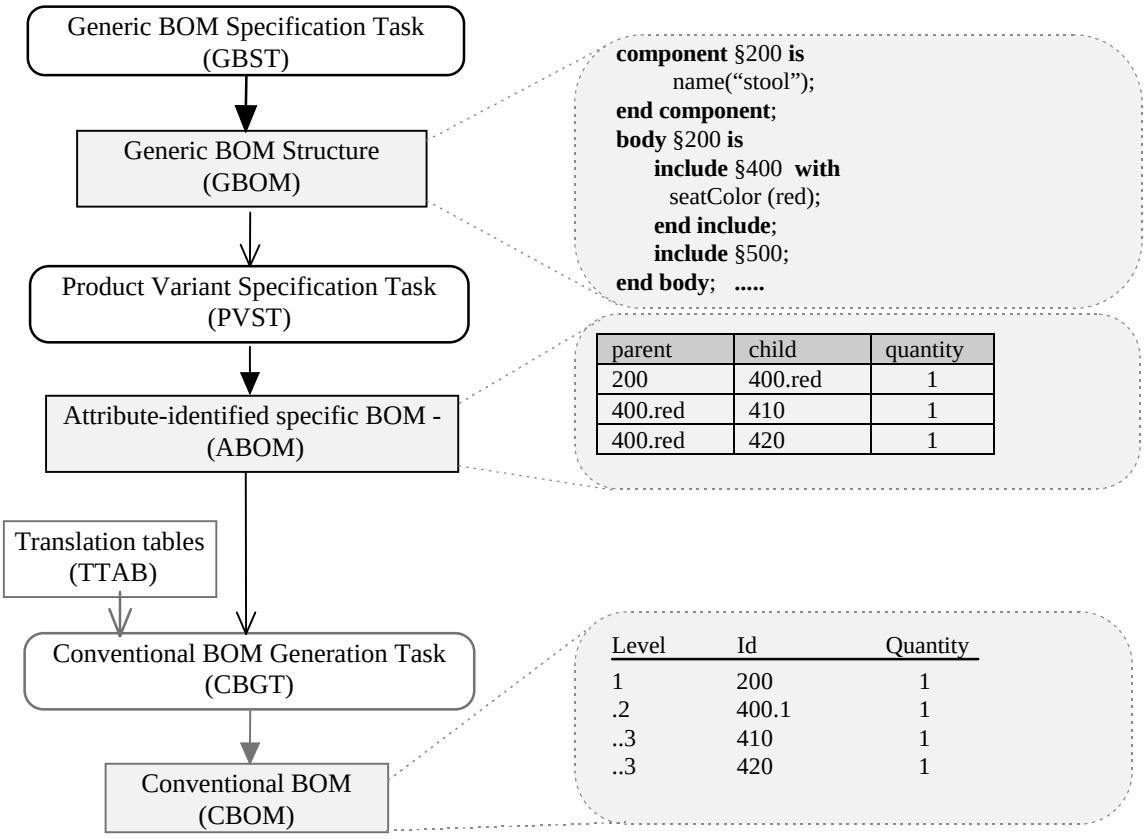

Figure 1. Generic system overview

The complete generic system is presented in Figure 1. It″ consists of three tasks, each task resulting in a data structure. Examples of the data structures are″ indicated. The tasks are:

1. The Generic BOM Specification Task (GBST) where a generic″ BOM structure for one or more products is specified.
2. The Product Variant Specification Task (PVST) where the″ attributes for a specific product variant are given. The result is a specific BOM for this″ product where a generic component is identified by its number, together with a value for each″ attribute.
3. The Conventional BOM Generation Task (CBGT). Each component″ referenced in the specific BOM with attributes is here replaced with a unique item″ number. The translation tables (TTAB) give the necessary conversion from attribute values to″ (sub) component numbers.

Task 3 is only necessary if the BOM is to be used by″ conventional systems that need a unique id-number for every component, that is for systems that cannot identify″ a variant by its attribute values.

The data structures, output from the above tasks, are:

1. The Generic BOM Structure (GBOM) describes top-level″ products as well as lower level components, in a manner as shown in the previous section. A″ GBOM will be complete when all referenced components are specified.
2. The Attribute-identified specific BOM (ABOM) is a″ description of a specific product variant. An ABOM is complete when the user has provided a specific value″ in all input situations.

3. The CBOM is a specific bill of materials, which may be used˝ directly in a conventional production control information system.

A typical scenario of utilizing the system will be as follows.˝ Production engineers with a complete knowledge of the products and production processes will˝ program the GBOM. The tool for entering the code may be as simple as a text editor or a more advanced *hypertext*-based system that may give direct links to products, components, parents, etc. However, we are˝ not limited to the text-based notation. An alternative is to allow a structured input of component˝ declarations in forms. Such an approach makes it easy to store each description as a structured data base˝ record, and will perhaps be more similar to what a production engineer would expect (see Section 5).

Independent of the choice of notation the GBST should˝ include a validation of the generic structure (e.g., all referenced components must exist, attributes must˝ be specified, values and value combination for attributes must be legal, there should be no cycles in˝ the structure).

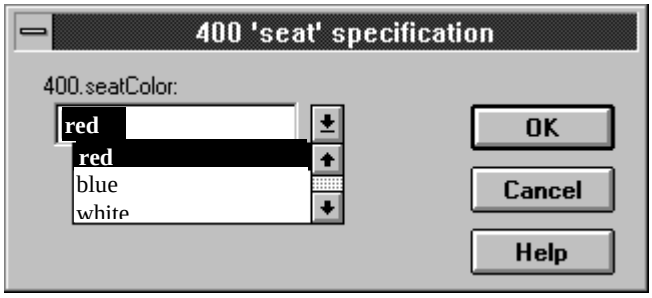

Figure 2. Product Specification (400 “seat”)

Sales people will use the GBOM in order to specify a product˝ variant. The PVST will be started by choosing a product from the generic structure. The product˝ structure is then executed (exploded). Generic components that have more than one possible value for˝ an attribute, present a value list to the user. This is seen from the example in Figure 2. Further˝ explosion is dependent on user input. An *undo*-feature is implemented through backtracking. In principle, no˝ expert knowledge of the product is needed in this phase, as the GBOM will control that the variant˝ specified is legal. For many products, the PVST may be performed with the customer on the phone. The result˝ of this task is a specific BOM with attribute identifiers (ABOM).

4. Constraints

In this section we have selected examples that have some of˝ the complexity of real world situations. Our product line will now be expanded with a typist chair. Both˝ stool and typist chair use the same seat, stand and base. Our earlier definition of seat has been extended˝ with a new attribute – texture. The stand comes in a fixed and a swivel variant, the base may have wheels or˝ feet. A typist chair has a plastic back and may have arm rests.

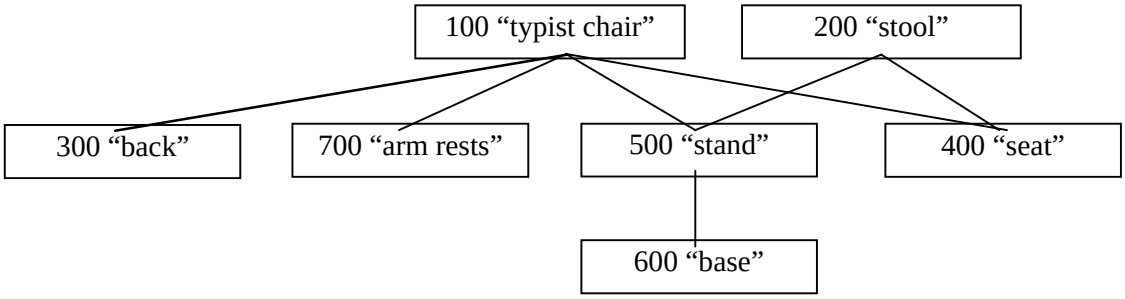

Figure 3. A generic product structure

The generic product structure is shown in Figure 3. We can˝ describe the main parts of the GBOM as follows:

```
component §100 is
  name("typist chair");
end component;
body §100 is
  include §300;
  include §400;
  include §500;
  include §700;
end body;

component §200 is
  name("stool");
end component;
body §200 is
  include §400;
  include §500;
end body;

component §300 is
  name("back");
end component;
body §300 is
  include §310;    -- back plate
  include §320;    -- frame
end body;

component §400 is
  name("seat");
  texture(wool|vinyl|leather);
  seatColor(red|blue|white);
end component;
body §400 is
  include §410;      -- cushion
  include §420;      -- chipboard
  case texture is
    when wool:
       include §450 with
          coverColor(seatColor);
       end include;
    when vinyl:
       .....
    when leather:
       .....
  end case;
end body;
```

```
component 450 is
  coverColor(seatColor);
  size(50x60);
end include;

component §500 is
  name("stand");
  type(swivel|fixed);
end component;
body §500 is
  case type is
    when swivel: include §501;
    when fixed:  include §502;
  end case;
  include §600;
end body;

component §600 is
  name("base");
  baseType(wheels|feet);
end component;
body §600 is
  include §620; -- support
  case baseType is
    when wheels:
       include §621 with
          quantity(5);
       end include;
    when feet:
       include §622 with
          quantity(5);
       end include;
  end case;
end body;

component §700 is
  name("arm rest");
  armRest(low|high);
end component;
body §700 is
  case armRest is
    when low:   include §701;
    when high:  include §702;
  end case;
end body;
```

The variant set of each component is described through the˝ component attributes, and the legal values of these attributes are indicated. The components are described˝ independently of their use in higher-level

products. We also note that there are no restrictions as to˝ which variants we may select for a given product, for example, both typist chair and stool may have˝ wheels.

Of course, this is a situation we seldom find in the real˝ world. We shall therefore introduce constraints to our structure. Additional features of our specification˝ language will be introduced through these examples.

Constraint 1: A stool must have feet

To achieve this constraint, stool must constrain a child of˝ its child, because the specification of feet is a part of base. We use the *constrain* statement to modify the body of stool:

```
body §200 is
   include §400;
   constrain §600 with
      baseType(feet);
      display("Feet required");
   end constrain;
   include §500;
end body;
```

When 600 (base) eventually is included as a component of˝ 500 (stand), the constraint will be in effect, thus reducing the choice of the baseType attribute to˝ "feet". Note that a component may constrain the attributes of *any* other component in the structure. The display statement˝ will present the given text for the user when component 600 is exploded. The effect of the˝ constrain statement will last until the end of the execution, or until the inverse statement – *unconstrain* – is executed.

Constraint 2: A swivel chair with wheels must have high arm˝ rests

To avoid damage claims from people falling off chairs, we˝ have decided to enforce high arm rests on a (typist) swivel chair with wheels. Note that this rule does˝ not apply to swivel chairs with feet, or to chairs with a fixed stand that have wheels. A modification of the˝ body of 600 (base) is then needed:

```
body §600 is
   include §620; -- support
   case baseType is
      when wheels:  ....;
         case §500's type is
            when swivel:  constrain §700 with
                             armRest(high);
                             display ("High arm rests required");
                          end constrain;
            when fixed:   ;
         end case;
      when feet:    ....;
   end case;
end body;
```

In the case statement we *reference* the type attribute of 500 (stand). Since this attribute˝ is declared in another component, it must be preceded (prefixed) by the˝ component identifier followed by a **'s**. The system employs a static data structure and all attributes may˝ be referenced in this manner. Attributes declared in components that are not yet executed will have˝ the value *undefined*. Thus, if the current value of the *type* attribute above is swivel, we execute the constrain and the˝ display statements, otherwise we take no action (a semicolon indicates a dummy statement).˝ When arm rests are included in typist chair (the "include §700" statement), the attribute˝ armRest may thus already have been specified.

Constraint 3: Version dependent declarations

A GBOM will never be static, products will be improved,˝ redesigned, produced out of different parts, etc. Some of these changes will go into effect immediately,˝ i.e., as the GBOM is updated. However, with major changes the update process itself may go over a long˝ period. In such cases it may be necessary to synchronize the effect of changes, such that all are realized˝ at the same time. We may control this by the use of a version number, for example, using the following˝ code in all components that are to be modified:

```
case §master 's versionNumber is
   when 1: include ...
   when 2: include ...
end case;
```

When the version number is changed from 1 to 2 the new˝ structure will be used instead of the old. The version number may be declared in a "master"˝ component:

```
component §master is
  versionNumber (1|2);
  ...
end component;
```

Such a component represents a convenient location for storing˝ global variables that are not natural component attributes. These global attributes may be assigned˝ values through the constrain statement, and may be referenced by use of the case statement (as seen˝ above).

5. Prototype implementation

A prototype of the generic BOM system has been implemented.˝ The prototype runs on a PC under MS Windows and consists of:

1. A tool for entering or editing the generic BOM structure˝ (the GBST).
2. A compiler that verifies the generic structure and˝ translates the user-oriented notation to an internal form (a set of tables).
3. A run-time system that allows the end user to execute˝ (explode) a component in a depth-first-order to specify a product variant. User specifications˝ (inputs) are given as the BOM is executed (the PVST). Output from this execution is a˝ specific BOM, where component variants are identified by attribute values (ABOM).
4. A translation of the ABOM to a conventional BOM (the˝ CBGT).

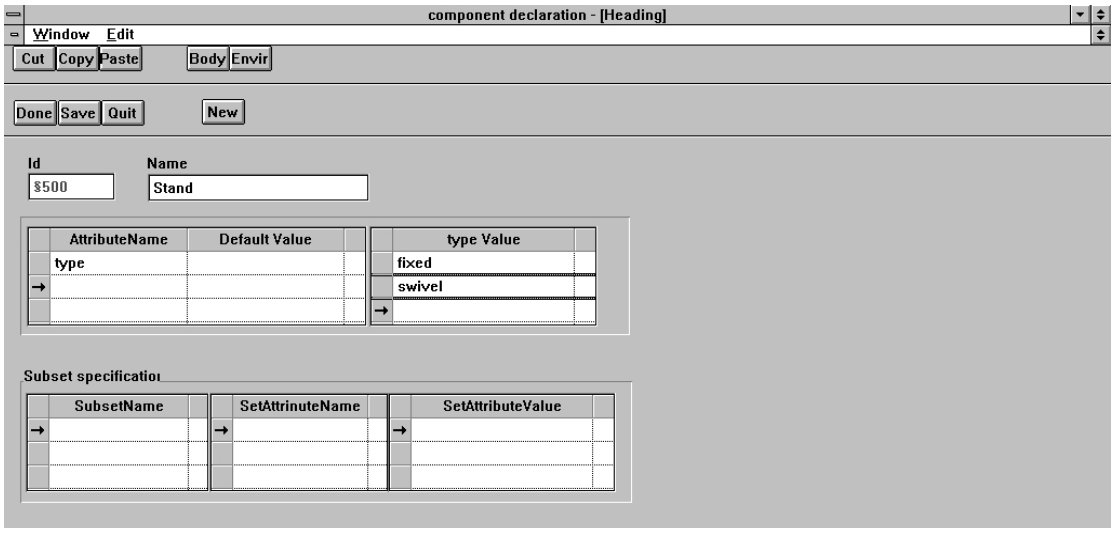

Figure 4. Form for describing a component head.

All programs are implemented with the GUPTA's SQL Windows″ case tool. Figure 4 shows the form used for describing the component head. The form offers fields″ for component identifier, name, attribute identifiers, legal values, etc.

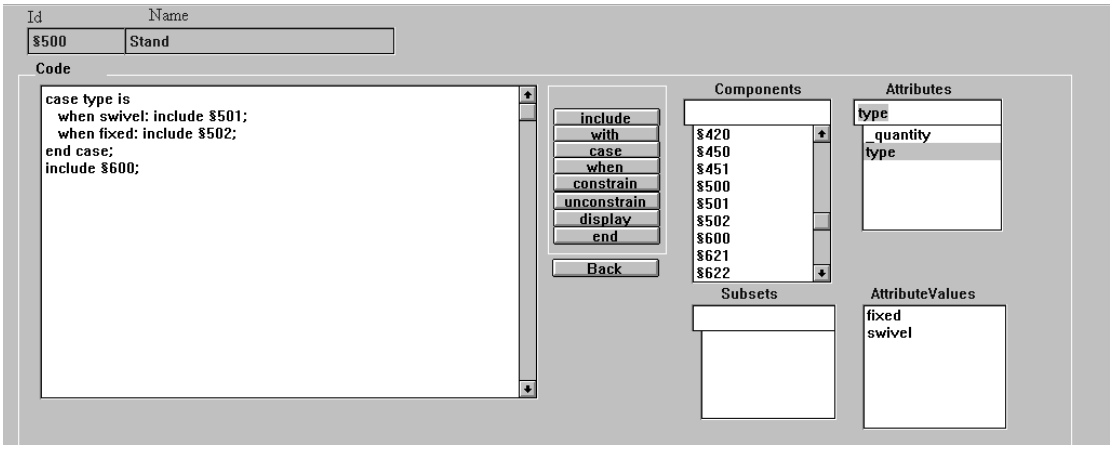

Figure 5. Form for entering the body part of a component.

The form for entering the body part is given in Figure 5. The″ buttons and list boxes to the right of the window offer a structured editor, i.e., reserved words,″ component identifiers, attributes and values may be entered by selecting from the appropriate lists. In this″ way, a syntactical correct body part can be constructed by use of the mouse only.

In addition to these two windows, an environmental window will show the components that include, constrain or reference a component.

6. Conclusions and future work

A generic BOM system has been introduced. A major part of″ this system is a programming language to describe generic product structures. This procedure-oriented″ approach gives a high degree of flexibility, and allows for the formulation of intricate constraints. At″ the same time it supports the order-entry task, i.e., the process where a user selects a specific product″ variant. A conventional BOM for such a variant may be generated based on the generic BOM and input from the″ user. A penalty of the procedure-based notation is that production engineers must learn a new tool″ for describing BOM's.

We have plans to further increase the power of the generic″ specification language by including macros (a named set of statements), arithmetic expressions and″ functions. These constructs give the possibility of defining global conversion rules for attributes, e.g., to″ construct a macro for converting from one standard to another or a function to calculate the right″ variant for a given specification. These extensions to our language will also allow for the specification of″ elaborate constraints. For example, we may calculate the total costs of options selected for a car and″ ensure that this is less than a given amount or assure that the power consumption of all the boards in a rack″ does not exceed the capacity of the power supply.

With these language extensions we may construct generic BOM″ structures that support organizational functions such as product development, forecasting and″ accounting. In fact, the procedure-oriented generic BOM will make it easier to combine the "as″ sold" and "as built" view of a BOM. A main task of our research is to study how the generic structure may be″ combined with prognostic and simulation techniques, to support planning and other management″ activities.

References

Blaha, M.R., Premerlani, W.J., Bender, A.R., Salemme, R.M.,˝ Kornfein, M.M.& Harkins, C.K. (1990) Bill-of-Material Configuration Generation, *Sixth International Conference on Data Engineering*, 237-244.

Bottema, A. & van der Tag, F. (1992) A Product Configurator˝ as Key Decision Support System, *IFIP transactions B, Applications in technology*, North-Holland, **B-7**, 71-92.

Chung, Y. & Fischer, G.W. (1992) Illustration of˝ object-oriented databases for the structure of a bill of materials, *Computers in Industry*, **19**, 257-270.

Chung, Y. & Fischer, G.W. (1994) A conceptual structure˝ and issues for an object-oriented bill of materials (BOM) data model, *Computers & Industrial Engineering*, **26**, 2, 321-339.

Hegge, H.M.H & Wortmann, J.C. (1991) Generic˝ bill-of-material: a new product model, *International Journal of Production Economics*, **23**, 117-128.

Schönsleben, P. (1985) *Flexible Produktionplanung und Steuerung mit dem Computer*, CW-Publikationen, München.

VanVeen, E.A. (1992) *Modelling product structures by generic bills-of-material*, Elsevier Science Publishers, Amsterdam.

VanVeen, E.A. & Wortmann, J.C. (1992a) Generative bill of˝ material processing systems, *Production Planning & Control*, **3**, 3, 314-316.

VanVeen, E.A. & Wortmann, J.C. (1992b) New developments in˝ generative bill of material processing systems, *Production Planning & Control*, **3**, 3, 327-335.

Wedekind, H. & Müller, T. (1981)˝ Stücklistenorganisation bei einer grossen Variantenanzahl, Angewandte Informatik, **9**, 377-383.

Appendix - Formal specification of the GBOM-language

An abbreviated formal specification of the language introduced˝ for describing the generic BOM structure is given below. The notation is based on the BNF˝ (Backhus-Naur-Form or Backhus-Normal-Form). Capital letters indicate a special property, for example,˝ COMPONENT_identifier is an identifier that refers to a component. Items in brackets ([ ]) may be˝ omitted, a bar (|) indicates an option and may be read as an *or*, while braces ({ }) indicate a repetetition (*one* or more times).

```
string ::= {letter|digit|special_character|space}

comment ::= -- string

identifier ::= [ § ] { letter | digit }

component_declaration ::= component_head [component_body]

component_head ::=  component COMPONENT_identifier is
                       [component_name]
                       [ {attribute_clause [:= DEFAULT_value_specification]} ]
                       [ {named_subset} ]
                    end component;

component_name ::= name ( " string " ) ;

attribute_clause ::= ATTRIBUTE_identifier ( value_list ) ;

value_list ::= VALUE_identifier [ { | VALUE_identifier } ]

named_subset ::=    subset SUBSET_identifier is
                       { attribute_clause }
                    end subset;

component_body ::=  body COMPONENT_identifier is
                       {statement}
                    end body;

statement := null_statement
   |  include_statement  |    case_statement
   |  display_statement  |    constrain_statement
   |  unconstrain_statement

null_statement ::= ;

display_statement ::= display ( " string " ) ;

constrain_statement ::=
   constrain COMPONENT_identifier [SUBSET_identifier] with
      { CONSTRAIN_attribute_clause | display_statement }
   end constrain;

unconstrain_statement ::=  unconstrain COMPONENT_identifier;

include_statement ::=
   include COMPONENT_identifier  [SUBSET_identifier] [with
      [ quantity ( NUMBER_identifier ) ; ]
      [ { CONSTRAIN_attribute_clause | display_statement } ]
   end include ] ;

case_statement ::=
   case [ COMPONENT_identifier 's ] ATTRIBUTE_identifier is
      { case_statement_alternative }
   end case;

case_statement_alternative ::= when value_list | undefined :  {statement};
```