In Proceedings of the 8th International Symposium on System Configuration Management (SCM-8), Brussels, July 20-21, 1998. LNCS-1439, pp 107-126, Springer-Verlag.

Product Configuration Using Object Oriented Grammars

Görel Hedin[1], Lennart Ohlsson[2], and John McKenna[3]

[1] Dept of Computer Science, Lund University, Sweden. Gorel.Hedin@dna.lth.se
[2] Utilia Consult, Malmö, Sweden. lennart@utilia.se
[3] Alfa Laval Thermal AB, Lund, Sweden. djohn.mckenna@alfalaval.com

Abstract. This paper presents a technique for product configuration modelling based on object-orientation and attribute grammars. The technique allows efficient interactive configurator tools to be generated for specified product families. Additional benefits include a high degree of checkability, early validation, readability, and reusability. The technique is particularly aimed at mass-customization products and an example concerning the mechanical configuration of plate heat exchangers is used to demonstrate its benefits.

1 Introduction

Complex products are often designed as product families where each customized product is a configuration of interrelated components. A *product configurator* is a tool which supports the product configuration process so that all the design and configuration rules which are expressed in a product configuration model are guaranteed to be satisfied. The configurator simplifies the manufacturing process by assuring that all orders received are possible to build. Interactive configurator tools can support quick and flexible customization by giving immediate and accurate information about the available combinations of options.

Ideally, the product configuration model should be written in a form which captures the design intent in a direct way, allowing the model to be easily read and changed by the domain engineers themselves, without involving expert programmers. It should be possible to generate computer-aided configuration tools automatically from the model. In this scenario, the software cost for a product change can be easily predicted, and changes can be introduced within days.

As a step towards this ideal scenario, we have developed OPG (Object-Oriented Product Grammar) - a configuration model description language which is based on object-oriented programming and attribute grammars. Object-orientation provides suitable modelling support and attribute grammars forms a suitable base for expressing configuration rules. Attribute grammars furthermore provide a well-established and efficient technology for the generation of interactive tools [16, 10].

The OPG work has grown out of an effort to formally describe the mechanical configuration of plate heat exchangers in a cooperation project between Lund University and Alfa Laval Thermal, the world leading supplier of plate and spiral heat exchangers. Alfa Laval Thermal has developed an interactive configuration tool called CAS 2000 (Computer Aided Sales) [1] which is used at market companies around the world. The tool allows customized design of plate heat exchangers. The user can interactively

enter requirements on e.g. pressures and temperatures, and select a number of different options according to engineering rules. The tool automatically presents valid alternatives for selection, and also checks that the designed plate heat exchanger is internally consistent, i.e., that it is possible to build. When a design is complete, the tool can prepare quotations for customers and send electronic orders to the appropriate manufacturing unit.

The immediate goal of the OPG work is to be able to replace the hand implemented configurator in CAS 2000 with a configurator generated automatically from declarative descriptions, thereby allowing the tool to be updated quickly to incorporate new product developments, eliminating the need for programming.

The rest of this paper is organized as follows. Section 2 gives an overview of our running example: plate heat exchangers, and the configuration problems involved for this product. Section 3 explains the architecture we propose for product configuration, and section 4 discusses key elements in the OPG formalism. A discussion is given in section 5 where we also give an outlook on future work. Section 6 discusses related work and section 7 concludes the paper.

2 Plate Heat Exchangers

A heat exchanger is a device for heating or cooling a fluid by exchanging heat with another fluid. Examples of use are in chemical processing, dairy processing, air conditioning, etc. A plate heat exchanger (PHE) consists of a pack of metal plates and the fluids are directed into alternate channels between the plates so that a large area for heat exchange is obtained without letting the fluids mix. Fig. 1 shows the basic structure of a PHE.

Our example concerns the *mechanical configuration* of the PHE, i.e. how a plate package is configured with frame plate, pressure plate, carrying bar, support column, tightening bolts, etc. Each frame plate and pressure plate has four holes which can be used in different ways. Four of the holes are used for connecting the incoming and out-

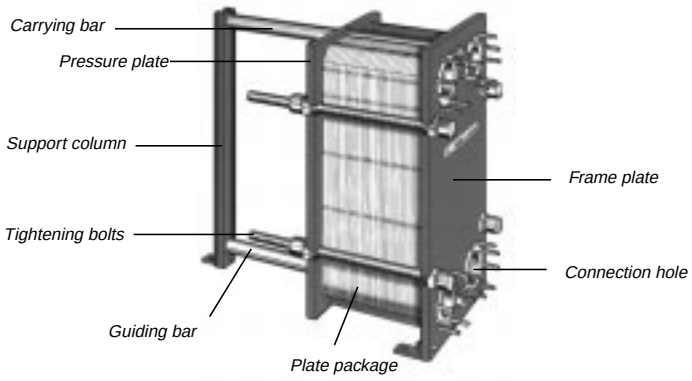

Fig. 1. Mechanical configuration of plate heat exchanger

going two fluids. Other holes can be covered with inspection covers (to allow inspection of a fluid). Some holes will not be in contact with any fluid, because the plate package does not have a hole in the corresponding position. In that case, the hole is simply covered by a blind cover. Examples of configuration options include how to place the connections and covers, how to select components of matching sizes and of appropriate material so that adjacent materials do not corrode and so that the components will stand the pressure requirements. When different configurations are possible, the cheapest one is usually preferable.

What is interesting about this example is that PHEs as such are not mass produced. Rather, the individual components, such as plates, carrying bars, support columns, etc., come in many different variants (each of which is mass produced), but each individual sold PHE usually represents a unique configuration of components. The components come in many different sizes and material in order to fit different plate package configurations and to cater for the customer requirements on for example pressure and corrosion properties.

Alfa Laval Thermal has for more than two decades applied leading edge production principles such as process oriented organization and build-to-order manufacturing, or what is now known as mass-customization. The product model used is known as a building block system. From a manufacturing perspective a block is a named assembly with a fixed set of components, i.e. it identifies a particular bill-of-material. For example, a given "tightening bolts block" identifies a certain set of actual nuts and bolts. From a sales perspective, however, a block appears as a component in the sense that blocks are the atomic elements which can be configured into a customer specific product.

3 The OPG Configuration Model

Taking a general perspective, we can view a product configuration as a set of connected *components*, each with a number of *properties*. A number of *rules* over the components and properties define the validity of the configuration: A *valid product configuration* is a configuration where all rules are satisfied. This general view is common in product configuration, see e.g. [19, 17]. An interactive product configurator tool, like CAS 2000 discussed above, allows the user to interactively edit a product configuration by adding, removing, or changing components. The tool constantly checks if the current configuration is valid and also helps the user to select components which will result in a valid configuration.

OPG (Object-Oriented Product Grammar) is a formal product configuration model based on object-orientation and attribute grammars. This model allows specification on three levels:

Type level: Specification of product types and their principle components, e.g. plate heat exchangers and their principle components such as frame plate, pressure plate, carrying bar, etc.

Prototype level: Specification of the mass-produced components which can be used in product configuration. E.g., specification of different mass-produced frame plates.

Configuration level: Specification of individual customized configurations. E.g., a milk cooler configured from instances of specific mass-produced components.

Our work aims at the possibility to automatically generate product-specific configurator tools like CAS 2000 by giving the product type specification as input to a tool generator and using the description of the mass-produced components as a database of the configurator tool. Fig. 2 outlines this approach. The main reason to support the generation of the configurator tool is to be able to quickly incorporate changes to the configurator as the product type evolves. In addition, it is important to support a high degree of consistency checking at all three levels, and that the specifications are highly readable and reusable to support evolution.

3.1 The Component Hierarchy

As mentioned, a product configuration is a set of connected components with properties and rules. In OPG, the components are connected in a hierarchy. In section 5 we will discuss the possibilities of generalizing this to a graph. A very simplified component hierarchy for a plate heat exchanger is shown in Fig. 3. The top component represents the complete PHE consisting of a PlatePackage and a Frame. The Frame in turn consists of a FramePlate, a PressurePlate, and TighteningBolts. This PHE does not have any CarryingBar or SupportColumn - these components are in general optional, but may be required in specific configurations. All four Holes on the FramePlate have contact with fluid (FluidContact) and are fitted with Linings. On the PressurePlate, three of the Holes are not in contact with fluid (NoFluidContact) and are covered with BlindCovers, whereas the fourth hole is in contact with fluid (FluidContact) and is fitted with a Lining and an InspectionCover. The PlatePackage has, in principle, an internal configuration of different kinds of plates, but since we are dealing with mechanical configuration only, we simply view it as a black box here.

Some of the components, e.g. FramePlate, correspond to physical mass-produced components, and we call these *prototypical components*. These components will typically have a bill-of-material associated with them. Other components do not have a physical correspondence and are called *configuration components*. These components may have the role of grouping prototypical components, e.g. Frame, indicating a con-

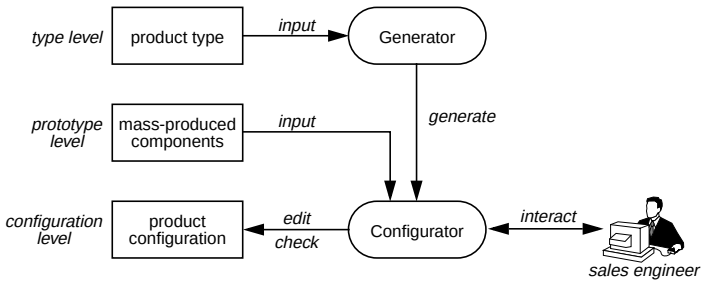

Fig. 2. Specification levels and tools

figuration choice, e.g. FluidContact and NoFluidContact, or modelling customer requirements, e.g. PHE.

A component *c* on the path towards the root from a component *s* is said to be *enclosing s*. Inversely, the *s* component is said to be a *subcomponent* of *c*. For example, the InspectionCover component in Fig. 3 has the following enclosing components: FluidContact, Hole, PressurePlate, Frame, PHE.

3.2 The Specification Levels

Different aspects of the components, connections, properties, and rules, are specified at different specification levels:

Type level: At this level component types are declared to model the types of components which occur in configurations. The types are arranged in an inheritance hierarchy of more general and more specific types (supertypes and subtypes). For each component type, its connections (in the component hierarchy) are declared, indicating mandatory and optional subcomponents, and arrays of subcomponents. Each component type also contains declarations of properties and rules.

Prototype level: At this level *prototypes*, i.e. instances of the prototypical component types, are specified, to model the components that are mass produced. Each prototype is given explicit values for some or all of the connections and properties declared in its type. Usually, all of the properties of a prototype are given values at this level, but the possibility to leave out some of the property values allows the

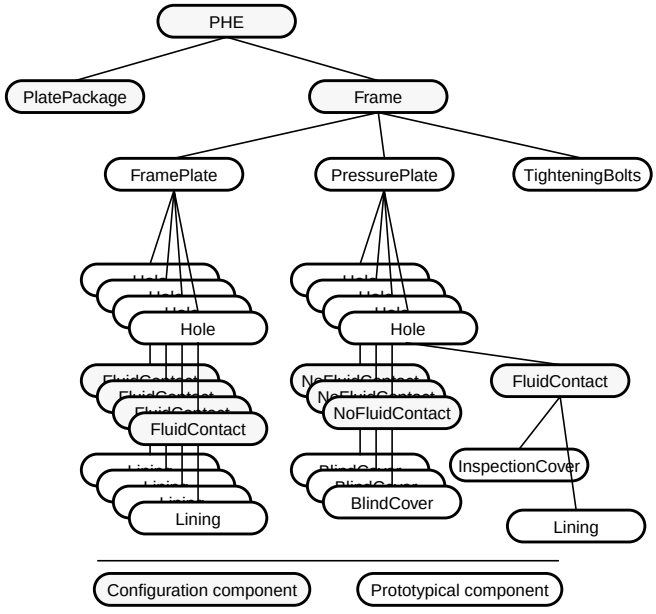

Fig. 3. A component hierarchy

specification of *parameterized* components. For example, an insulation board may be manufactured as piece goods, sold by the metre, and its property *length* may be given a value at the configuration level rather than at the prototype level.

Configuration level: At this level instances of configuration component types and copies of prototypes are specified and the (remaining) connections and properties are given values.

Fig. 4 shows an example of types, prototypes, and a partial configuration. The type level specifies prototypical types like FramePlate, Hole, and Lining, and configuration types like HoleContact and its two subtypes FluidContact and NoFluidContact. At the prototype level, different prototypes for FramePlate (FP1, FP2, ...) and Lining (L1, L2, ...) are specified with differing property values. The example also shows the possibility for prototypes to specify subcomponents, by specifying the different Holes for a FramePlate. At the configuration level, instances of FluidContact are specified and connected with copies of the FP1 and L1 prototypes.

The model is clearly heavily inspired by object-orientation. The component types correspond to classes, the prototypes to prototype instances of the classes, and the instances at the configuration level to instances of classes or copies of prototypes. The object-oriented concept of prototypes and copies [13] has also been used in the area of product configuration by e.g. Peltonen et al. [15]. Our use of prototypes is, however, a direct application of the Prototype design pattern [7] rather than the use of prototype or delegation-based programming as in [13, 15].

The model is also heavily inspired by attribute grammars. An attribute grammar is an extension of a context-free grammar with attributes and semantic rules [12]. OPG component types with its connections correspond to the nonterminals and productions of the context-free part of the grammar, and the OPG properties and rules to the attributes and semantic rules of the grammar. Some of our previous work [8] shows the benefits of using object-oriented notations for attribute grammars, similar to how it is done here. The configuration, i.e. the component hierarchy, corresponds to an attributed syntax tree, and the notion of valid product configuration is directly analogous to the notion of a syntax tree with a valid attribution. Attribute grammars have proven especially suitable for generating interactive language-based editors [16], and our earlier work shows how to use object-oriented attribute grammar techniques to obtain very efficient incremental evaluators [10]. Such evaluators can incrementally check the attribution validity as the user edits the syntax tree, and the technique is directly applicable to product configurator tools where an automated mechanism can check the validity of a configuration as it is edited.

An important aspect of product configurator tools is that they should support the user not only with checking validity, but with selecting valid components. To provide attribute-grammar based support for this is an area for future work, but we have developed fragments of such techniques in the area of language-based editing, where e.g. semantic editing can be used to provide a user with menus of visible and type-correct identifiers [9].

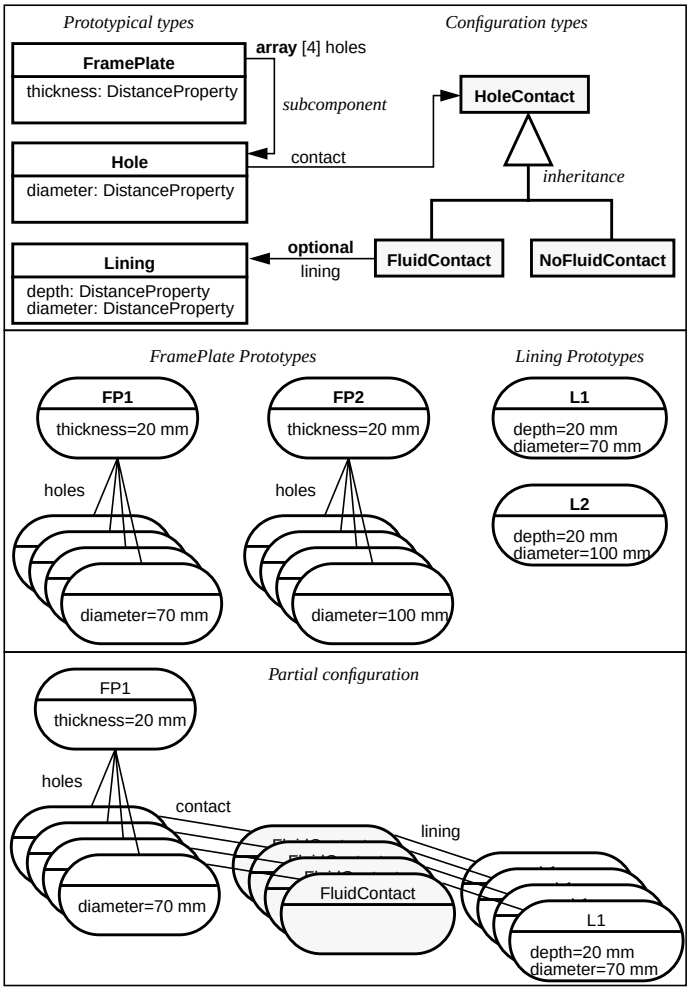

Fig. 4. Examples of types, prototypes, and instances in a partial configuration

4 Elements in OPG

4.1 Properties

All properties are declared at the type level, but may be given values at the prototype or configuration level. There are three different modes for properties depending on how they are given values: prototype properties, configuration properties, and derived properties.

Prototype properties. These are properties which are given values at the prototype level. A prototype property typically reflects some physical characteristic of the com-

ponent, e.g. thickness or diameter as in the example above, or some imposed characteristic such as a price.

Configuration properties. These are properties which are given values at the configuration level, either by the user explicitly or computed by some external tool. A configuration property set by the user typically reflects a customer requirement. For example, the Frame type has several configuration properties such as operatingPressure, upperTemperature, and lowerTemperature. External tools may also set configuration properties, allowing these tools to be interfaced to the main configurator. For example, an external tool computing the plate package configuration could set the properties of the PlatePackage component, e.g. length, plateMaterial, and gasketMaterial. These properties can then be used by the main configurator to check the validity of the mechanical configuration.

Derived properties. The value of a derived property is computed by a rule, making use of other property values or constants. Typically, derived properties are introduced in order to make other rules simpler to express. We will see examples of this later. There are two principle ways of defining the value of a derived property: by a rule in the component holding the derived property, or by a rule in the immediately enclosing component. This can be used to propagate property values up or down a component hierarchy in order to make them easily accessible at different locations. A derived property does not have to be explicitly stored, but can be computed whenever its value is needed. It can thus be implemented as a function.
The prototype and configuration properties correspond to lexemes in a context free grammar (e.g. identifiers, integer constants, boolean constants, string constants, etc.). Derived properties, defined upwards or downwards, correspond exactly to the synthesized and inherited attributes in an attribute grammar. As discussed in [8], it is particularly simple to implement the attributes (derived properties) as functions by using the virtual function construct in object-oriented languages.

4.2 Rules

Rules in OPG are placed in the component types and are specified at the type level. A rule is local in the sense that it can refer to the properties of *self* (an instance of the component type) and also to other components accessible in certain ways from *self*. There are two kinds of rules: defining rules and validity rules:

Defining rules: A defining rule is a rule defining the value of a derived property. It corresponds to a semantic rule of an attribute grammar.

Validity rules: A validity rule is a boolean expression over properties. A configuration is said to be valid if all the validity rules in the configuration are satisfied. A validity rule corresponds to a semantic condition in an attribute grammar.

OPG offers two ways of accessing properties in other components than *self*:

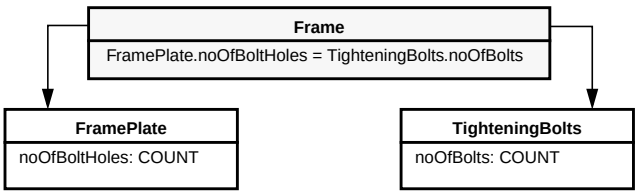

Fig. 5. Rule accessing subcomponent properties

Subcomponent reference: A rule may access or define properties of immediate subcomponents of *self*.

Enclosing reference: A rule may access a property of an enclosing component qualified by a given type.

Fig. 5 shows an example of accessing subcomponent properties: A validity rule in Frame checks that the FramePlate subcomponent has the same number of bolt holes as there are bolts in the TighteningBolts subcomponent.[1]
The possibility to access and define properties in *self* and in immediate subcomponents corresponds directly to the mechanism for defining and accessing attributes in classical attribute grammars. In principle, this possibility is sufficient for expressing arbitrary rules, because it is always possible to introduce derived properties which copy the information from one point in the configuration hierarchy to another. However, this introduction of derived copy properties can be cumbersome and lead to a cluttered specification. This problem is well known in attribute grammars and many attribute grammar notation languages therefore introduce various shortcut notations to avoid having to introduce such copy properties, see e.g. [11]. The *enclosing* mechanism mentioned above is one such shortcut notation (called "including" in [11]). By letting a component *c* directly access a property *p* in an enclosing component *e*, we can avoid introducing a number of derived properties which copy the value of *p* from *e* down to *c*. Besides cutting down on the number of properties and rules, this shortcut notation is very important for reusability reasons because it very effectively reduces the dependencies between different component types. In the next section we will see how the *enclosing* mechanism can be used very effectively in combination with mixin types.

In general, if a rule needs to refer to properties in two components c_1 and c_2, there are different alternatives for where to place the rule. By using a combination of derived properties and the *enclosing* mechanism, the rule could be placed in any of the components on the path from c_1 to c_2. Which placement is chosen can greatly affect the size, readability, and reusability of the specification.

As an example: consider checking that the depth of a Lining is the same as the depth of the Hole in which it is placed. In our example, Hole has no explicit property *depth*, but if the Hole is in a FramePlate, the depth of the Hole is of course the same as

[1] In case a component has only one subcomponent of a particular type, the type name (e.g. FramePlate) can be used to access the subcomponent. It is also possible to give subcomponents explicit names as was done in Fig. 4.

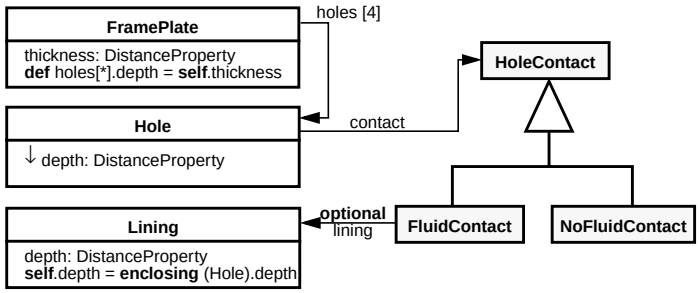

Fig. 6. Combined use of derived properties and the *enclosing* mechanism

the thickness of the FramePlate. Fig. 6 shows a solution which combines the use of derived properties and the *enclosing* mechanism.

In this example, *depth* is defined as a (downwards) derived property (indicated by "↓"). The FramePlate has a defining rule which defines the value of *depth* of the Hole subcomponents. The Lining has a validation rule which uses the *enclosing* mechanism to access the *depth* property and compare it with its own *depth* property. It would be possible to obtain a slightly smaller specification by letting the Lining refer directly to the thickness property of FramePlate, thus making the depth property and its defining rule unnecessary. However, that would make the specification less reusable because it would not work for Holes in other components like, for example, PressurePlate. This could be fixed by refactoring the type hierarchy, introducing a supertype for FramePlate and PressurePlate where the thickness property would be declared, but this would on the other hand make the specification bigger. We also find the solution in Fig. 6 more readable: the two rules express closely our explanation of the problem in informal english.

4.3 Main Types and Mixins

The component types we have seen so far model prototypical components and configuration components. We call these *main types* because the type captures the main aspect of the component. Main types can be organized in a single inheritance hierarchy to model different levels of generality. For example, (see fig. 7) BlindCover models blind covers at a general level, whereas its subtypes LargeBlindCover and SmallBlindCover are specialized alternatives: a LargeBlindCover must stand the pressure imposed from the plate package and is therefore bolted to the FramePlate or PressurePlate, whereas a SmallBlindCover is attached by a simple snap mechanism. This is modelled by specifying different properties and rules in LargeBlindCover and SmallBlindCover.

It is often the case that component types which are unrelated in the main type hierarchy, nevertheless share some properties and rules. In this case, one would like to express the shared behavior in a separate type and reuse it by inheriting that type to the appropriate main types (by multiple inheritance). Such types are called *mixins*, see e.g. [4]. OPG mixin and main types differ in the following ways:

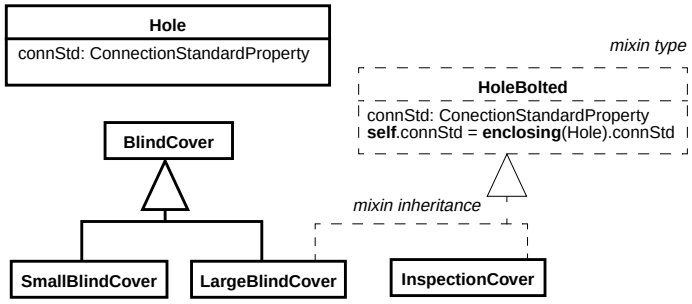

Fig. 7. Use of mixin types

- Whereas a main type models the main aspect of a component, a mixin type models only a partial behavior which may be common to several unrelated main types.
- A main type may inherit from at most one other main type (single inheritance), but from any number of mixin types (multiple inheritance).
- Inheritance of mixin types is done in order to obtain specification reuse whereas inheritance of main types is done to model specialized alternatives in a configuration.
- A subcomponent must be declared to be of a main type. Mixin types may not be used.
- A mixin type may not contain declarations of subcomponents, only property declarations and rules.
- A mixin type is always abstract, i.e. it cannot be instantiated on its own.

The separation of types into two kinds: a full kind (main types) which can be arranged using single inheritance, and a partial kind (mixin types) which can be arranged using multiple inheritance is often recommended in object-oriented programming, see e.g. [20]. This substantially improves readability and reusability over general multiple inheritance, while retaining most advantages. A similar technique is also used in Java with its separation into class types and interface types [2]. The combination of mixins with shortcut notations like “enclosing” is very powerful. In attribute grammars this technique has been used very effectively to define e.g. reusable scope rules [11] (the use of mixins is there called “symbol inheritance”).

Fig. 7 shows an example of main and mixin inheritance. LargeBlindCover and InspectionCover are unrelated in the main type hierarchy, but nevertheless have a common characteristic: they are components which are bolted to a hole in a FramePlate or PressurePlate, and they need to use the same connection standard as is used for the hole in the plate they are bolted to. To avoid repeating the specification of this characteristic we specify it in a mixin type HoleBolted which is inherited by both LargeBlindCover and InspectionCover.

Pairs of Mixin Types

The use of the *enclosing* construct makes a component dependent on some type in its context. If not used with care, this may lead to reduced reusability, in case it is meaningful to use the component in some other context. Therefore, when a component type *C* refers to an enclosing type *E*, the following question should be posed: would it be meaningful to have a *C* component in some other context which does not contain an *E* component? If the answer is "yes", the reusability of *C* is unnecessarily limited because of its dependency on *E*. In our examples of *enclosing* above, the enclosing type is Hole. It seems reasonable to argue that the accessing types Lining, InspectionCover, and LargeBlindCover are meaningful only in the context of a Hole, and there is thus no problem with reusability.

In other cases, when the answer is "yes", one may consider introducing a mixin type also for the enclosing component. This introduces a pair of mixins which are designed to work together: the "upper" mixin declares some properties which are accessed using the *enclosing* mechanism in the "lower" mixin. This solves the reusability problem because it makes the mixin types independent of the actual components where they are mixed in, allowing the behavior to be mixed in for different components and different contexts.

Consider the following example. In an operating PHE, the fluids have a certain pressure, and many of the components like FramePlates, PressurePlates, InspectionCovers, and LargeBlindCovers must stand at least this pressure. A first solution to this problem would be to introduce a mixin type PressureClassified which declares a property maxPressure and uses the *enclosing* mechanism to compared it with the operatingPressure property of PHE. However, it is quite possible that we would like to reuse the InspectionCover and LargeBlindCover types when modelling another product, for example a spiral heat exchanger (SHE). In that case, these types would appear in the context of a SHE component, and the maxPressure should instead be compared with the operatingPressure property of SHE. The solution is to introduce a mixin PressureVessel which is inherited by both PHE and SHE as shown in Fig. 8. The mixins PressureVessel and PressureClassified operate as a pair. They model the general behav-

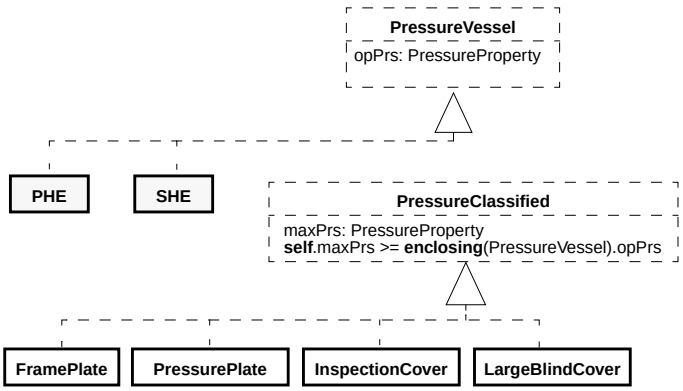

Fig. 8. Use of mixin pairs

ior of having a pressure vessel with an operating pressure, and components attached to it are classified to stand at least that pressure. The mixin pair can then be reused for any product exhibiting this general behavior.

5 Discussion

OPG is aimed at obtaining descriptions which are small, highly readable, and easy to evolve and reuse; to obtain a high degree of checkability; and to allow automatic generation of efficient interactive configuration tools. We will now discuss different aspects of OPG and how it contributes to these goals.

Architecture. The proposed architecture which is divided into the three levels of types, prototypes, and configuration, supports the different phases in the design and configuration process. Product designers define a family of related products by working at the type level. The prototype level describes the different mass-produced components. The type level serves as an interface for defining new prototypes, placing requirements on what physical and other external characteristics the new prototype must possess. If values can be set for these properties, it will be possible to define the prototype and use it in future configurations. Typically, these prototypical properties will serve as requirements on subcontractors manufacturing the mass-produced parts. Sales engineers work at the configuration level, defining how actual customized products are configured from mass-fabricated parts. This architecture fits mass-customization problems like e.g. plate heat exchangers. For other kinds of products the design may be more intertwined with configuration, e.g. as indicated in [15].

Checkability. The three-level architecture allows a clear definition of what kinds of validity can be checked at each level. At the type level, the type system itself allows a basic validity check, similar to the compile-time checking of strongly typed programming languages like Pascal and Java, e.g. making sure that all accessed properties actually exist (i.e. that they are declared). The AG-inspired use of derived properties furthermore allows checking if the derived properties are well-defined, i.e., that for any possible configuration, there will be exactly one defining rule for each derived property, thus avoiding under- or overdetermination with regards to derived properties. This is an important aspect because it supports early error checking and can make designers more confident they have designed a consistent model.

At the prototype level, it is possible to check if the prototype is complete, i.e. if all prototypical properties are given values. It is also possible to do a partial check on the validity of the prototype, by checking validity rules which use only prototypical properties of the (compound) prototype. At the configuration level, it is possible to do full validity checking by checking if all validity rules are satisfied.

Interactive Configurator Tools. As described earlier, the AG technology allows efficient interactive configurator tools to be generated from the type-level specification. Such tools can support structure-oriented editing of configurations (using graphical or textual editing techniques) and can check the validity as the configuration is edited,

using incremental attribute evaluation techniques. The validity checking can be done very efficiently by AG evaluation techniques, making use of statically computed evaluation plans (tables computed at tool generation time). It is important that the incremental evaluation is efficient in order to allow an interactive construction of the configuration, giving immediate feedback on possible rule violations.

Readability and Reusability. All rules in OPG are local to a component and may access properties of other components relative to that component. This is in contrast to the global "for all" rules which often are used in knowledge-based systems. Because of the implicit identification of the component itself, subcomponents, and components in its context, local rules are usually much easier to understand. As discussed in section 4.3, a high degree of reusability is obtained through the combined use of mixins and the *enclosing* mechanism.

5.1 Future work

We have used OPG successfully to model the mechanical configuration of plate heat exchangers, resulting in a specification of 26 main types, 7 mixins, and 32 rules. Some aspects which are not supported currently in OPG, but which are interesting topics for future research include the following:

Support for Valid Choices. The AG-based model immediately supports only *checking* the validity of a configuration. It is also desirable to support the user in *constructing* a valid configuration. The user builds a configuration by successively adding components, and at any time, the current partial configuration will limit the valid choices for remaining components. The configurator should have support for constructing valid configurations by presenting only the valid choices, and for automatically adding components when there is only one valid choice. It is also possible to automatically complete a partial configuration using default components and configuration properties. We plan to add such support by formalizing the choice-generation framework used in CAS 2000 and integrate it into the OPG model.

Support for Graphs. OPG supports only configurations organized as a tree of components. In general, one would like to support also graphs. We plan to support this in a similar way as is done in Door attribute grammars [10], i.e., by introducing reference-valued derived properties. This allows an arbitrary connection between two of the components in a tree to be set up using derived properties, thus in effect turning the tree into a graph. Rules can then access properties directly along such a connection.

Support for Versioning. A product family is usually not constant, but evolves over time and the descriptions should be subject to revision control. It is very important that existing type-instance and prototype-copy relations are not made inconsistent when the types and prototypes evolve. These problems are similar to schema evolution problems in object-oriented data bases, see e.g. [21], and similar techniques will probably be useful for product models.

6 Related Work

6.1 Product Configurator Tools

Table-Based Tools. Many simpler hand-coded configurator tools use a table-based approach. In these systems, each base product has a table which lists its available features. A product is specified by choosing a base product and then adding a number of features. When two features are not allowed to be combined for some base product, this combination is listed in another table, a conflict table. This approach is based on the assumption that features are highly independent of one another so that conflicts are rare. At Alfa Laval Thermal, this technique was used in the first version of CAS (CAS 1), developed in 1988. A drawback with the table based approach is that it can handle only quite simple dependencies between product components. To allow more complex dependencies it is necessary to have some kind of rule concept which allows configuration rules to be expressed over the components and their properties.

Framework-Based Tools. Another approach to configuration systems is the *object-oriented framework* approach. Here the components of a configuration are modelled in an object-oriented manner. All configuration rules are described as matching conditions between attributes of the objects, and the logic is implemented by change propagation rules. By ensuring that these rules define uni-directional chains, a highly efficient change propagation mechanism can be implemented in the generic framework. The actual rules for a specific product are then expressed by extending the framework with hand-written code. As a result the model is fairly maintainable although there is a translation step from domain experts to programmers which limits verifiability. This approach is used in Alfa Laval's current CAS 2000 system developed in 1992 [1].

Knowledge-Based Tools. We differ between simpler *rule-based* systems and more advanced *constraint-based* systems. *Rule-based* configuration systems are based on logic programming and use a technology similar to expert systems. The formalism allows all the relevant rules to be stated explicitly, and the order of evaluation is decided by a general purpose inference engine which thereby determines the execution logic. Rule-based systems are declarative, but they often have only limited support for modularization. Typically, they work in a non-interactive mode, computing a complete configuration from user requirements. An example of a rule based configuration system is the seminal XCON (or R1) system for configuration of computer systems [14].

An evolution of the rule-based systems are the *constraint-based* systems. Constraint-based systems typically have two major advantages over conventional rule-based systems. Firstly, different kinds of resolution strategies can be applied to different kinds of constraints which can give significant improvements in execution efficiency. Secondly, constraints are usually defined in some kind of object-oriented model of the product. The classes in an object-oriented model gives a natural basis for modularization so that large models can be structured in a way that make them locally understandable. Constraint-based configurators thereby achieve both declarative

expression of configuration logic and a natural modularization of the model. Recent constraint-based tools such as OBELICS [3] also support interactive configuration, allowing the user to interactively select key components.

Comparison. A weakness of knowledge-based systems is that they are based on a dynamically determined execution order. For hard configuration problems, i.e., when the configurator has to optimize over a large space of possible configurations, the flexibility and possibilities for global optimizations outweighs the drawbacks. Many configuration problems, however, are not hard. What is important is instead that the tool can be highly interactive and give immediate feedback on what the user selects. In other words, the system should be highly supportive of running "what-if" scenarios.

OPG's attribute grammar approach presented in this paper combines the advantages of declarative statement of optimized execution efficiency and highly responsive interactive support. As the user changes the configuration, dependent rules can be re-evaluated incrementally according to a statically determined execution order, giving immediate feedback on the validity of the configuration.

OPG is also focused on providing early error detection in the product model. This is supported by the strong type system and in the possibility to check that all attributes are uniquely defined by the rules. This is in contrast to the tradition in knowledge-based systems which are usually based on dynamic checking only.

6.2 Product Data Management

Product configuration modelling is one aspect in the larger context of Product Data Management (PDM) which covers all information related to product design and manufacturing. Most PDM system in industrial use today have little functionality that is specific to product data. Their main emphasis is versioning of files (typically for CAD-programs and word-processors) combined with work flow support and change management. Usually some form of support for the bill-of-material concept, i.e.hierarchical configurations, is also included.

PDM is slowly evolving away from this document-centric view towards more explicit product models. In a product model, information is structured so that it can be easily processed by different kinds of software tools, for example product configurators. When needed, various kinds of documents can be derived automatically from the model.

Product modelling technology is often based on some kind of object-oriented formalism with mechanisms for describing properties and rules. The most notable example here is the EXPRESS language [17] created by the STEP initiative which is an ongoing effort to create an international standard for the exchange of product model data.

On the surface, OPG is similar to EXPRESS because they are both product modelling languages with constructs like classes, properties, and rules (called entities, attributes, and rules in EXPRESS). However, there are many differences, the main one being that OPG explicitly supports an attribute grammar model with a component hierarchy (abstract syntax tree) and upwards and downwards derived properties.

EXPRESS supports upwards derived properties, but downwards derived properties and the use of properties in enclosing classes would have to be simulated by using validity rules and explicit specification of enclosing components, leading to complex specifications not suited for AG processing and less suited for reuse and early error-detection. Furthermore, OPG has explicit support for prototypes whereas in EXPRESS these would have to be simulated by subclasses. To summarize, OPG is a product modelling language suitable for mass-customization products and for the generation of interactive configuration tools. It would, however, be possible to translate OPG to EXPRESS in order to make OPG specifications available to EXPRESS-based tools.

6.3 Software Configuration Management

Configuration management techniques have been developed independently for mechanical products and software products, and it may be interesting to compare these different problems and techniques.

In both cases, there is a configuration problem, i.e. a problem of selecting specific components to form a complete product which is internally consistent in some way. An overview of Software Configuration Management (SCM) systems is given in [5]. Most SCM systems are *version-oriented*, where each component exists in several versions organized as *variants* (alternative versions) and *revisions* (consecutive versions). Often, the component set making up the product is predefined, e.g. in a makefile, and the central configuration problem is to select a suitable version for each of the components, typically in order to configurate a product suitable for a given execution platform. In OPG, the central configuration problem is instead to select which components make up the product.

A fundamental difference is that a component in OPG may occur in several instances in the product, e.g. the PHE configuration in Fig. 3 contains 4 instances of Lining. In SCM, on the other hand, a component appears at most once in the configuration. In some SCM systems, the definition of the component set is intertwined with version selection, resulting in a configuration process more similar to OPG. However, the fundamental difference concerning multiple/single component instances remains.

The types and prototypes in OPG may be compared with components and variants in SCM: prototypes may be seen as different variants of their type. In OPG, the prototypes of a type differ in their property values. Similarly, the variants of a component in SCM are often characterized by attributes or features [6, 22]. However, the selection rules work quite differently. In OPG, the rules are localized to the components and check consistency between a component and its enclosing components and subcomponents, e.g. checking that a Lining has the same diameter as its enclosing Hole. In SCM, the rules are usually global and used to select component variants regardless of their context, e.g., selecting the Unix variant of all source module components in the system. One SCM system which does make use of local rules is DCDL [18] where rules are part of class definitions and can check consistency between a component and its subcomponents.

Revision control is central in SCM: within a component variant, there may be several consecutive revisions which can be attributed with date, release number, etc. In

contrast, each OPG variant (prototype) exists in only one version. The reason is that the physical components modelled by prototypes are not seen as being in a revision relation: either they are completely interchangeable, in which case they are modelled by the same prototype; or they differ in some property values, in which case they are modelled by different prototypes. Typically, different physical components are modelled by the same prototype if they are manufactured by different subcontractors, but have the same function in the product, and which actual component is selected at manufacturing is irrelevant to the customer. Revision control is, however, highly relevant to OPG at its *meta level*, i.e. when specifying OPG types and prototypes in order to generate a configuration tool. These specifications are evolving software which can be placed under revision control, as noted in section 5.1.

The current trends in software architecture of using component technology will give rise to new configuration problems in software which have similarities to the mass-customization problems treated by OPG. As interconnection standards like COM, CORBA, and Java Beans are coming into wide use, the granularity of the units of deployment is decreasing. Monolithic applications are giving way to a larger number of more or less independent components, moving the main configuration problem from build-time to install-time, or even to launch- or run-time. The responsibility for building correct configurations is thereby separated from component development. This separation must then be compensated by the components being more self-contained. When delivered, they must contain sufficiently rich meta-data to enable automatic generation and/or validation of correct configurations. The problem is similar to that of product mass-customization, and it is possible that product configuration formalisms like OPG can play a role here. The SCM system DCDL [18] is aimed at run-time configuration problems, and as noted above it has some similarity to OPG in that it also supports local rules.

7 Conclusions

OPG is a product modelling language which is primarily aimed at configuration of highly customized products built from mass-produced components. We believe it shows the benefits of basing configuration technology on object-orientation and attribute grammars. This combination gives strong modelling capabilities, allowing configuration constraints to be expressed and understood locally. The *enclosing* mechanism and the use of mixin pairs in particular allows the model to be highly factorized so that redundant information is avoided. The type-prototype-copy architecture is introduced to match the different levels of product component type, mass-produced component, and actual component. The use of attribute grammars gives OPG both a theoretical basis for early validation and techniques for the automatic generation of efficient interactive configurator tools.

References

1. Alfa Laval Thermal AB. *CAS 2000. User's Manual*. Lund, Sweden, 1993.

2. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley. 1996.
3. T. Axling and S. Haridi. A tool for developing interactive configuration applications. *Journal of Logic Programming* 26(2): 147-168 (1996).
4. G. Bracha and W. Cook. Mixin-Based Inheritance. OOPSLA/ECOOP'90. *ACM SIGPLAN Notices*, Vol. 25, No 10, pp. 303-311. 1990.
5. R. Conradi and B. Westfechtel. Configuring Versioned Software Products. In *Software Configuration Management, ICSE'96 SCM-6 Workshop*. pp 88-109. LNCS 1167, Springer-Verlag. 1996.
6. J. Estublier and R. Casallas. The Adele Configuration Manager. In Tichy (Ed.) *Configuration Management*, Wiley, 1994.
7. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.
8. G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329-345, Cambridge University Press. 1989.
9. G. Hedin. Context-sensitive editing in Orm. Proceedings of the *Nordic Workshop on Programming Environment Research*. Tampere University of Technology, Finland. Software Syst. Lab. TR 14. 1992.
10. G. Hedin. An overview of Door attribute grammars. *International Conference on Compiler Construction (CC'94)*. LNCS 786, Springer Verlag. 1994.
11. U. Kastens and W. M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31:601-627, 1994.
12. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127-145, June 1968.
13. H. Lieberman. Using prototype objects to implement shared behavior in object oriented systems. In OOPSLA'86, pp 214-223. *ACM SIGPLAN Notices*, Vol. 21, No. 11, September 1986.
14. J. McDermott. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence*, Vol. 19, 1 (Sept 1982):39-88.
15. H. Peltonen, T. Männistö, K. Alho, R. Sulonen. Product Configurations - An Application for Prototype Object Approach. In *ECOOP'94*, pp 513-534. LNCS 821, Springer Verlag. 1994.
16. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator. A system for constructing language-based editors*. Springer Verlag. 1989.
17. D. Schenck and P. Wilson. *Information Modeling the EXPRESS Way*. Oxford University Press. 1994.
18. B. R. Schmerl and C. D. Marlin. Versioning and consistency for dynamically composed configurations. In *Software Configuration Management, ICSE'97 SCM-7 Workshop*. pp 49-65. LNCS 1235, Springer-Verlag. 1997.
19. J. J. Shah and M. Mäntylä. *Parametric and Feature-Based CAD/CAM*. Wiley. 1995.
20. Taligent Inc. *Taligent's guide to designing programs - well-mannered object-oriented design in C++*. Addison-Wesley. 1994.
21. A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In OOPSLA'86, pp 483-495. *ACM SIGPLAN Notices*, Vol. 21, No. 11, September 1986.
22. A. Zeller and G. Snelting. Handling Version Sets Through Feature Logic. In *Software Engineering - ESEC'95*. pp 191-204. LNCS 989. Springer-Verlag. 1995.