Product Configurations—
An Application for Prototype Object Approach[1]

Hannu Peltonen, Tomi Männistö, Kari Alho and Reijo Sulonen
Department of Computer Science
Helsinki University of Technology
Otakaari 1, FIN-02150 Espoo, Finland
{Hannu.Peltonen, Tomi.Mannisto, Kari.Alho, Reijo.Sulonen}@hut.fi

Abstract. Product configuration management is presented as a practical application for a prototype-based object model. Data model requirements for a configuration system are first introduced using a realistic example from industry. Problems with the traditional type-instance model in this application domain are then identified and given as motivation for the prototype approach. A prototype-based object model with inheritance tree transformations, constraints and component relationships is presented as a tool for expressing dynamic configuration data. Finally, a sample configuration process is described using the prototype object model.

Keywords: Prototype object model, delegation, product configurations.

1 Introduction

Most object-based models are built upon the concepts of object types (or classes) and their instances. In fact, sometimes the distinction between types and instances is presented as a fundamental property of object models. Nevertheless, object models based on prototyping (which typically use delegation for sharing information between objects) do not contain the concept of an object type [9, 11, 12].

In this paper we discuss a real-life application in which the prototype approach seems to have a clear advantage over more traditional models based on the type-instance distinction. The application—the management of product configurations—is introduced in Sect. 2.

Section 3 shows the problems of the type-instance model in our application and gives motivation for the prototype approach. Readers unfamiliar with prototypes should not have any trouble in following this section. For more background on prototype models, see for example [9].

---

1. This paper has appeared in the Object-Oriented Programming, M. Tokoro and R. Pareschi (Eds.), Proc. of the 8th ECOOP Conference, Lecture Notes in Computer Science 821, Springer-Verlag, pages 513–534, 1994.

We are developing a prototype-based object model; this model is described to some detail in Sect. 4. The object model specifies operations for manipulating data in the objects (represented as attributes) and conditions for valid data (represented as constraints). The objects, however, do not contain methods or other means to describe user-defined behaviour. Using the terminology defined by Wegner [18] the model described in this paper can be classified as *prototypical*.

Section 5 shows how the object model can be used to represent configurations and the steps in which the configurations are produced. Finally, Sect. 6 provides conclusions and briefly brings up issues that could not be treated in this paper.

2 Configuration Problem

This section outlines some main issues of product configuration management. We describe the problem domain before our solution because our research aims at solving a configuration problem; we are not looking for an application for the prototype object model.

In many industrial areas there is a constantly increasing demand for more customized solutions. The increased customization inherently forces some of the design decisions to be made only after the customer order has been received. Typically for this type of products it is not rational to design each single customer order from scratch. Instead, the customer needs should be fulfilled using combinations of predefined, usually parametrized components. Related portion of the product life cycle, i.e., the configuration process, is a routine design task for determining a consistent combination of components and their parameter values. Since this process should produce not only valid, but at least close to optimal configurations, mechanisms are needed for guiding the search for such solutions.

In certain cases it may be possible to consider all the possible solutions and execute a domain independent search through the design space, but in general that is not a feasible approach. Instead, one can design a product model, which has some of its properties or components fixed or restricted to certain choices and some left unrestricted because of technical, business related, etc. decision criteria. The configuration process then uses this domain specific knowledge for generating a suitable configuration [16].

Some product models allow a configuration to be created automatically from the customer specification. In these cases the products are totally pre-engineered and the essence of the models is to select certain component combinations with suitable attribute values.

Sometimes, however, the customer specification cannot be satisfied by any available product model. There are several ways of creating such non-standard configurations: (1) Starting from a clean table and selecting the components on the basis of designer's experience, (2) taking an old product as a starting point and making the needed component changes, or (3) starting from an existing product model and then at a certain point of time departing the design from the limitations of that product model.

In the first case, the configuration must still obey some general rules of the application domain; the configuration process thus never starts from a completely clean table. In the second and the third case, the constraints of the starting point (whether an old product or an existing product model) should be made available to the new design. Then the designer must only make those modifications that are absolutely necessary, reusing most of the existing knowledge.

The data model used by a configurator should describe the pieces of knowledge within the entities they belong to. Ideally, for example, components themselves describe their properties and constraints from their own point of view, whereas a product model describes how these components are used in a particular situation. This is essential since the total amount of the configuration knowledge of a complex product is very large, making its maintenance in a non-structured fashion impossible. In general the maintenance of large knowledge bases is identified as a major problem [2]—the problem is even amplified within the constantly evolving environment of industrial products. In addition to the changes and refinement of the individual configurations, we must thus also cope with the orthogonal evolution of the configuration meta information, i.e., product models, components, standards, etc.

Superficially the relation between product models and configuration looks similar to the relation between schema and data in the database world. This may suggest that schema maintenance methods for databases would also be successful in configuration meta information management. There are, however, two fundamental differences. Firstly, conversion of data after the schema is modified is generally out of the question, since the configurations may represent separate, physical entities possibly delivered to the customers. Changes in product models thus should not automatically propagate into these entities. Further, the configuration information needs to be accessible even years later, so we cannot sacrifice the representation function, i.e., the correspondence between entities in the database and the real world, by storing only the single current schema. Secondly, the configuration process includes operations, e.g., adding an attribute, that may actually be data manipulation in configuration domain but would be considered as data definition manipulation in the context of databases. These points will be further illustrated later.

A good domain for product configuration modelling research is provided by lifts since they vary from standard to semi- and fully customized [6, 10, 19]. Therefore, a model that defines the entities, operations, and processes for a lift configurator should also be applicable to a wide variety of other products.

3 Motivation for Prototype Approach

In this section we first describe briefly the properties of the traditional type-instance model, and then show what kind problems we encountered when we tried to apply this model to the configuration problem. Similar ideas, although in a less precise application domain, have also been presented by Demaid and Zucker [3].

The goal of this paper is to demonstrate the unsuitability of the distinction between types and instances in our particular application. The last qualification is important: We do not claim that prototype models would in general be better than type-based models (*object-based* models with classes and inheritance are called *object-oriented* by Wegner [18]).

3.1 Type-instance Model

The "traditional" type-based object model makes a sharp distinction between a schema, which describes the structure of the data, and objects, which store the actual data as described by the schema.

The schema defines a number of types, and each type contains a number of attribute declarations. Each object is an instance of a type and can assign values to attributes declared by its type. Figure 1 shows how product models could be represented as types and configurations as their instances.

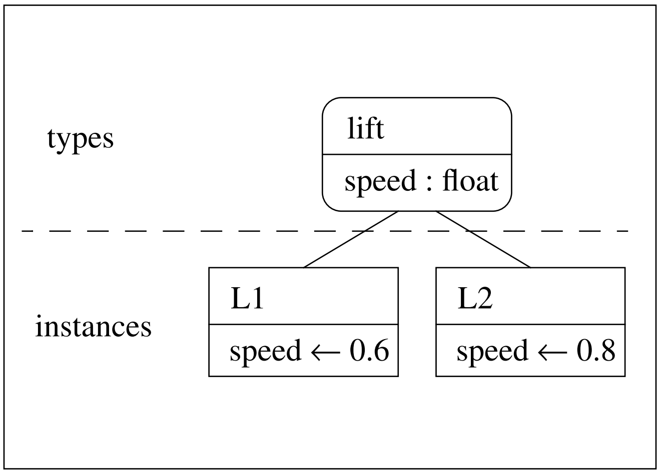


Figure 1: Types and instances

Note that this description applies equally well to relational databases, where a table corresponds to an object type and the rows of a table represent instances of that type.

Moreover, in most object-based systems, and in some extended relational systems (e.g., Postgres [13]), all types in a schema are organized as an inheritance hierarchy. As shown in Fig. 2, an instance of a type can assign values to all attributes declared by the type and its ancestors.

Figures 1 and 2 present types and their instances as a single tree. As long as one travels down the nodes in the "type world", each node adds new attribute declarations. When one crosses the border and enters the "instance world", one can have only a single node, which contains assignments to those attributes whose declarations were encountered on the "type" path.

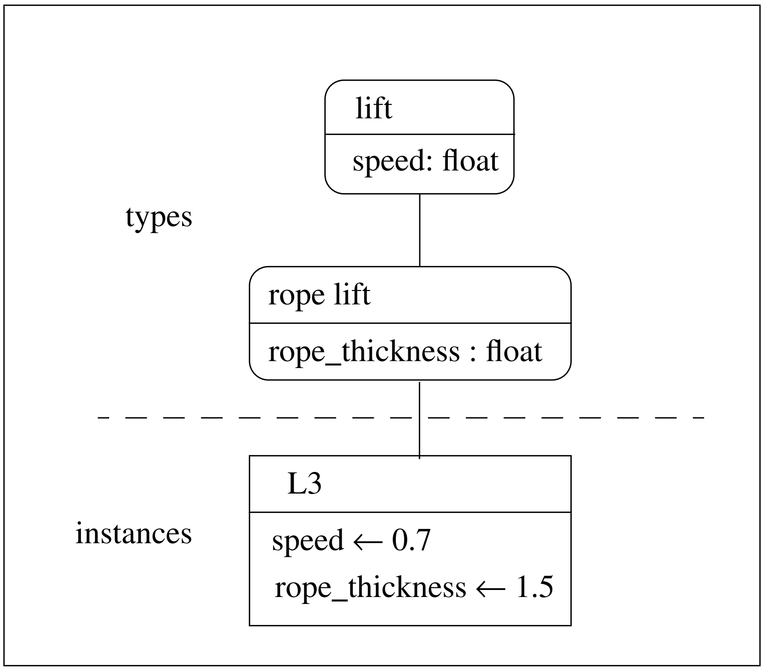


Figure 2: Type hierarchy and instances

The difference between types and instances also affects user roles. Types and their attributes are typically specified by a database administrator when the database is created. Ordinary users operate in the “instance world” by creating and deleting instances and modifying their attribute values. Types are modified only by the database administrator. Type modification is usually a major operation, undertaken only when absolutely necessary because of new requirements for the database system [1].

Often there are separate data definition and data manipulation languages. These terms reflect the static nature of the schema from an ordinary user’s point of view; the creation of a new type or a new attribute is not part of ordinary data manipulation.

3.2 Representation of Configuration Data

Figure 3 shows what kind of configuration data we want to represent with the object model. All boxes in the figure are rectangles because no distinction between types and instances is made. In fact, the figure will be used for demonstrating the impossibility of this distinction.

In addition to attribute declarations and assignments, the objects contain constraints written in curly braces. Constraints are conditions that must be satisfied by valid configurations.

The figure shows the following information:

- All lifts have a floating point attribute for the speed.

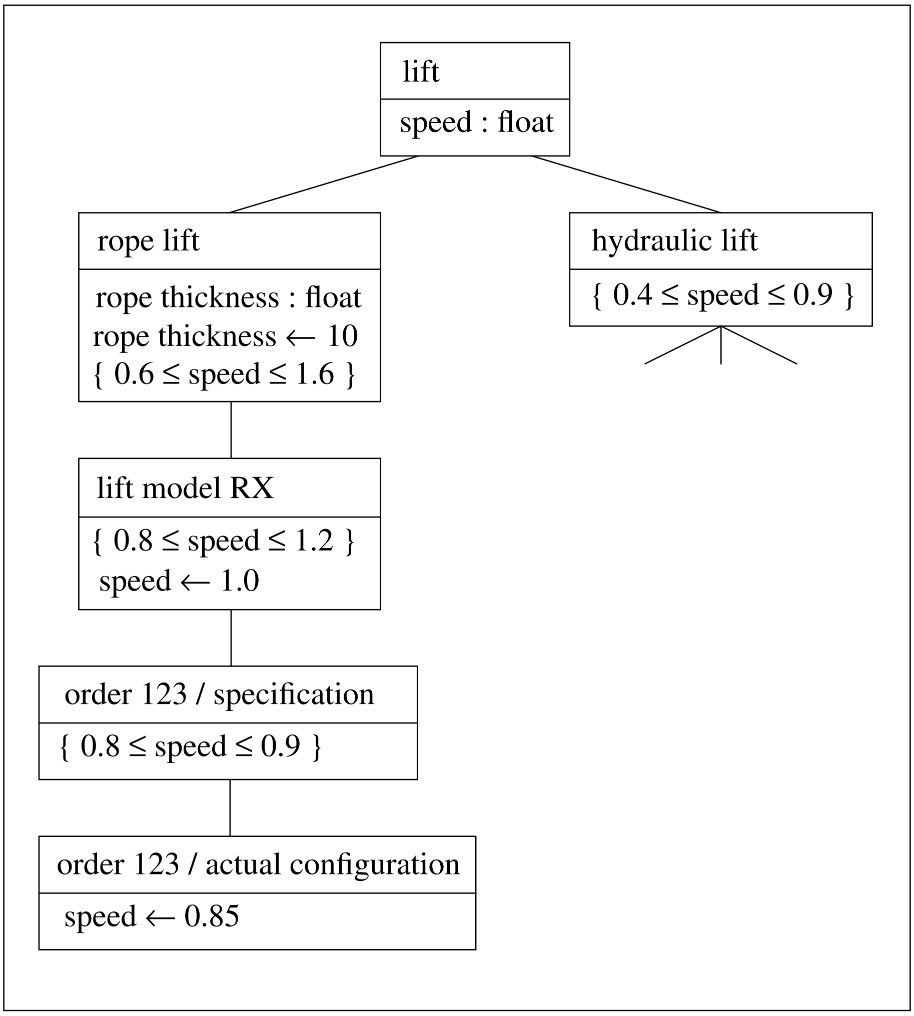


Figure 3: Configuration data

- A rope lift is a special case of a lift. Rope lifts have an attribute for rope thickness. The default value for this attribute is 10 mm. The speed of a rope lift is always between 0.6 and 1.6 m/s.
- A hydraulic lift is also one kind of a lift with speed between 0.4 and 0.9 m/s.
- Lift model RX is a rope lift with speed between 0.8 and 1.2 m/s. Default speed is 1.0 m/s.
- Customer order 123 is an order for a RX lift. The speed of the lift is specified by the customer to be between 0.8 and 0.9 m/s.
- The actual configuration for order 123 has speed of 0.85 m/s. Rope thickness has the default value 10 mm of all rope lifts.

The only way to force Fig. 3 into the type-instance model would be to regard the bottom-most entity (*order 123 / configuration*) as an instance and all other entities as types. This, however, would be awkward for the following reasons:

(1) The manipulation of an order is part of an ordinary user's job. The type-instance model, however, would regard the specification of an order as a type and the actual configuration as an instance. As pointed out in Sect. 3.1, most systems make a sharp distinction between operations on types and instances. It is much more natural to treat the two order 123 objects together as an "instance" and the definition of lift model RX as part of the "schema". This division of an order into separate objects will be further illustrated in Sect. 5.

(2) It may become necessary to add attribute declarations and constraints to "instances". (This is elaborated below.)

(3) It may become necessary to add new objects below "instances". (This is also explained below.)

For point (2) above, suppose the lift of order 123 will be installed in a hot environment in a factory and the engine room must be equipped with an extra fan, which is not specified by the lift model RX. The designer can represent this requirement by adding a new attribute declaration to the configuration object. As long as a specific fan model has not been selected, the configuration contains an attribute without an assigned value and the configuration system knows the configuration to be incomplete.

For point (3), suppose the designer has fixed the speed to be 0.85 m/s and wants to represent the fact that the speed must not be changed while other attributes, which are not shown in the figure, can be modified more freely. This can be done by creating a new object below the "instance" as illustrated in Fig. 4. Now attributes can be modified freely in the "open configuration" without the possibility of accidentally modifying attributes of the "fixed configuration". It is also possible that fixed configuration is created by one designer, who determines some important attribute values and gives the configuration to another designer for further work. This mechanism will be further elaborated in Sect. 5. The two separate objects can represent this division of responsibilities between the two designers. Note that the fixed configuration has both an assignment and a constraint for the speed; without the constraint the assignment would only be a default value that could be overridden in the open configuration.

4 Object Model Elaboration

The previous section presented our reasons for abandoning the division between types and instances in favour of a prototype-based object model. This section describes our elaboration of the prototype model in more detail.

All components, standard products and configurations are represented as objects. Each objects has its own identity, a name and *properties*. The properties comprise *attribute declarations* (Sect. 4.2), *attribute assignments* (Sect. 4.3) and *constraints* (Sect. 4.4).

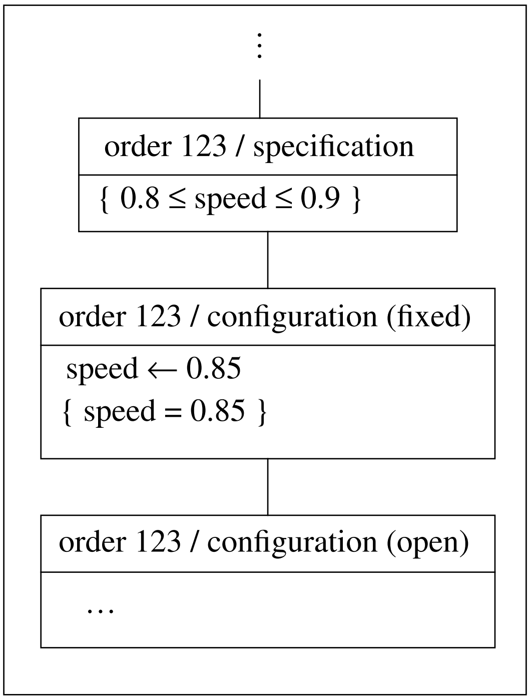


Figure 4: Fixing attribute values of a configuration

4.1 Object Inheritance

Each object, except the predefined *root object*, has a single object as the *parent object*. The parent object must be specified when a new object is created. As will be explained in Sect. 4.5, an object can be changed to have a new parent and this operation plays an important role in the representation of a configuration process.

The relationships *child*, *ancestor* and *descendant* are defined in the normal way from the parent relationship. As an object cannot be its own ancestor, the inheritance hierarchy forms a tree.

An object *inherits* all properties (declarations, assignments and constraints) of its ancestors. Moreover, an object is said to possess all properties that it either contains or inherits.

In the future the objects will be allowed to have multiple parents. Multiple inheritance might, for instance, represent country-specific safety regulations. Figure 5 shows lift *models X* and *Y*; safety regulations for two countries; and *model X* lift *order 234* for *Finland*. For brevity we have referred to the objects with names *safety regulations* and *Finland*, although *order 234* is not a kind of Finland, neither is Finland a kind of safety regulation. More precisely, object *safety regulations* represents lifts that satisfy certain safety regulations, and *Finland* represents lifts that fulfil the Finnish safety regulations.

Except for some specific examples, the rest of the paper deals only with single inheritance. Some additional uses for multiple inheritance will be given in Sect. 6.

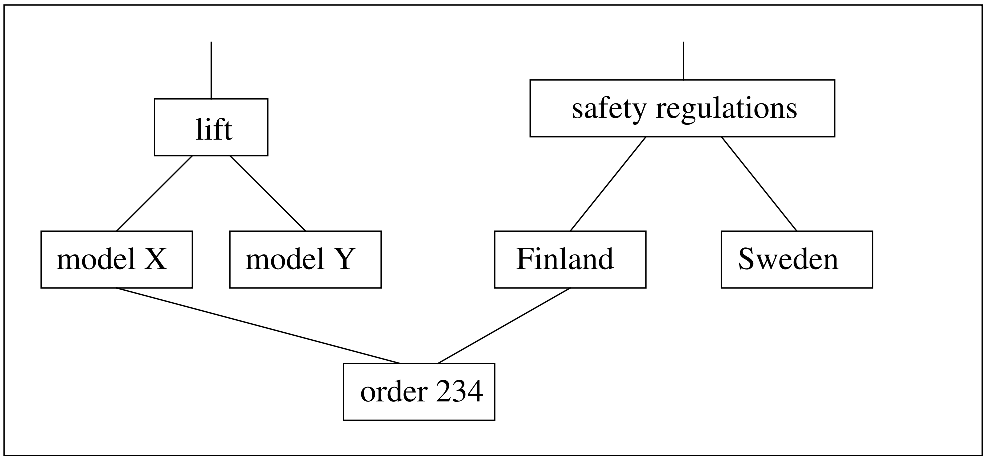


Figure 5: Safety regulations with multiple inheritance

4.2 Attribute Declarations

Objects store data as attributes, which must be *declared* before use. An object can therefore contain a number of *attribute declarations*. Each declaration specifies the name and type of an attribute. Simple attribute types include integers, floating point numbers and strings. Attributes that refer to other objects will be introduced in Sect. 4.6.

A newly created object does not contain any properties. An attribute declaration can be added to any object provided that the object does not already contain a declaration of an attribute with the same name. Note that an object and its ancestors can declare attributes with the same name (see also Footnote 2).

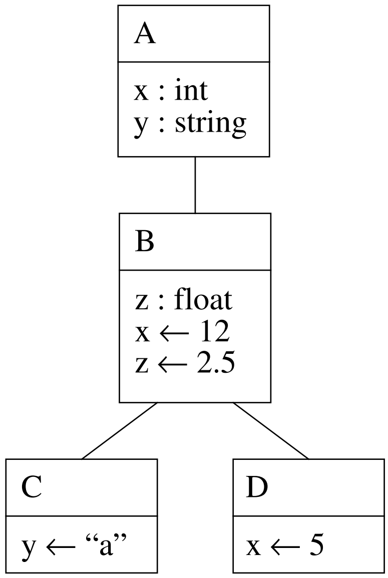
4.3 Attribute Assignments

If an object possesses the declaration of an attribute, an *attribute assignment* to the attribute can be added to the object (unless the object already contains an assignment to the same attribute). The assignment specifies the name[2] of an attribute and a value, which must conform to the attribute type in the declaration. The value in an assignment can be changed later.

When the value of a particular attribute in a particular object is needed, the object and its ancestors are examined from the object towards the root, and the attribute value is taken from the first assignment—the effective assignment—to the attribute. If only a declaration for the attribute is found, the attribute value is unknown. If not even a declaration is found, an attempt to read the attribute value is an error.

2. The name is only used for locating the attribute declaration. The actual assignment in the object does not store the attribute name but a reference to the declaration. The same applies to attribute references in constraints. Full treatment of this issue is beyond the scope of this paper.

Figure 6 shows four objects with attribute declarations and assignments. The lines between objects represent the parent-child relationships. The results of reading attribute values in the objects are shown on the right. Question marks denote unknown values.



Object	Attribute values		
	x	y	z
A	?	?	error
B	12	?	2.5
C	12	“a”	2.5
D	5	?	2.5

Figure 6: Attribute declarations and assignments

The value assigned to an attribute in an object can thus be regarded as a default value for the attribute in the descendant objects; the assignment is inherited, but any descendant can add an assignment to the same attribute, overriding the default value.

4.4 Constraints

In addition to attribute declarations and assignments, an object can possess *constraints*, which specify the conditions for the validity of the object. A constraint is an expression which evaluates to *true*, *false*, or *unknown*.

Basically, an object is valid if all the constraints it possesses are true, the object is *invalid* if any constraint is false, and otherwise the validity of the object is unknown. This approach to constraints and validity is similar to EXPRESS [5] language defined in the STEP programme.

The complete set of constraints of an object can be unsatisfiable. A system implementing the model may, but need not, detect this and tell the user that the object can never be valid.

4.5 Tree Transformations

As explained in Sect. 4.1, each object has a single other object as a parent. The model gains much of its flexibility from the possibility of changing an object to have a new parent. Many important operations, such as object copying, can be implemented by means of parent change.

The parent of an object can be changed freely as long as one does not attempt to create a cycle in the object hierarchy. The effects of a parent change depend on the relationship between the original and the new parent.

The inheritance rules for declarations, assignments and constraints mean that all descendants of an object have at least the same attributes as the object (but not necessarily the same attribute values), and all valid descendants of an object satisfy the constraints of the object. One can thus view an object as a description for a set of possible valid descendant objects. Each of these descendants in turn describes a subset of this set. (This bears some resemblance to the concept of derived subtypes in some semantic data models [4]. A derived subtype specifies a predicate, and all instances of the supertype that satisfy the predicate are automatically classified as instances of the derived type.)

Specialization. When the new parent of an object is a descendant of the old parent, the object is "specialized". It inherits more attributes and constraints than before and represents a smaller set of possible valid objects.

In Fig. 7, a lift order is represented with the object *order 345*. Originally the order is only specified to be some kind of a hydraulic lift. Later during the configuration process, the designer selects HEX from the available standard lifts as a basis for the order. This standard lift includes an automatic ventilation system and accordingly object *HEX* contains a declaration for attribute fan power. After *order 345* is changed to have *HEX* as the parent, an assignment to *fan power* can be added to *order 345*.

Generalization. An object is generalized by changing it to have a new parent which is an ancestor of the original parent. As a result, the object will inherit fewer properties than before. If an object is regarded as an instance of the type represented by its parent, the generalization of an object makes it an instance of a more general type.

We want the generalized object to possess the same attributes and constraints as before the operation. Therefore the intervening properties (declarations, constraints and effective assignments) are copied to the object. The generalization of an object does not change any attribute values or validity constraints. However, some attributes and constraints become "local properties" that can be modified without affecting other objects (except of course the descendants of the generalized object). This approach differs from [1] where the removal of a class from the superclass list of a class requires dropping existing instance variables. The possibility of making local definitions for attributes as a result of inheritance changes is mentioned in [20].

Figure 8 shows the same objects as Fig. 7. Lift has a new attribute *max load*, which is constrained in the standard lift HEX to lie between 100 and 500. The maximum load of

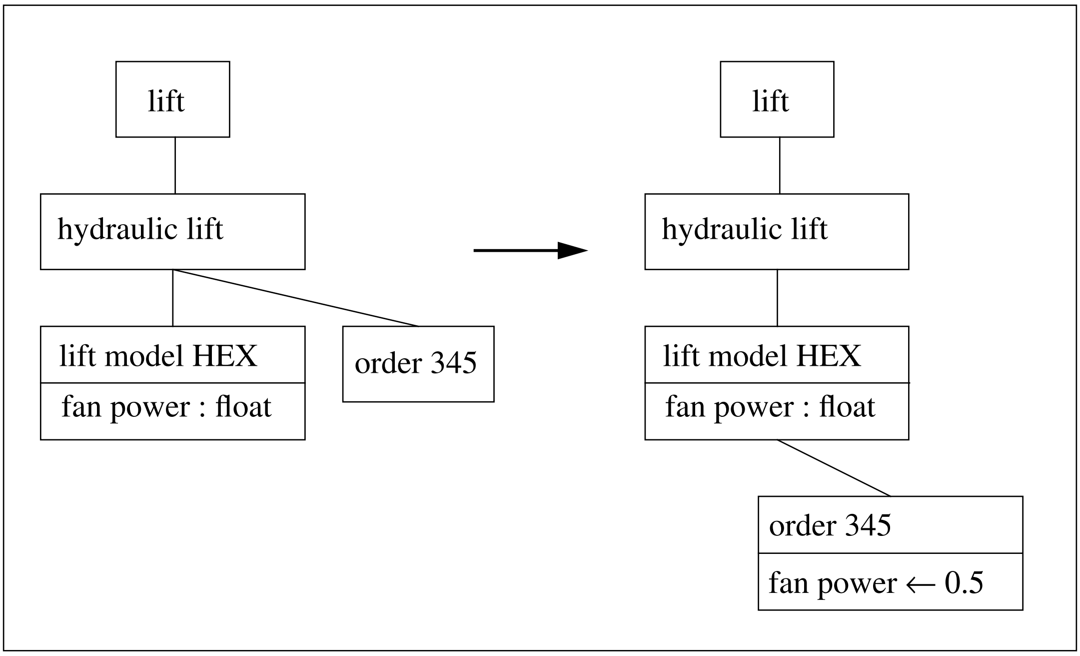


Figure 7: Object specialization

order 345 has been assigned value 600. Since this violates the inherited constraint, the order no longer belongs to the set of valid lifts as specified by the standard lift HEX.

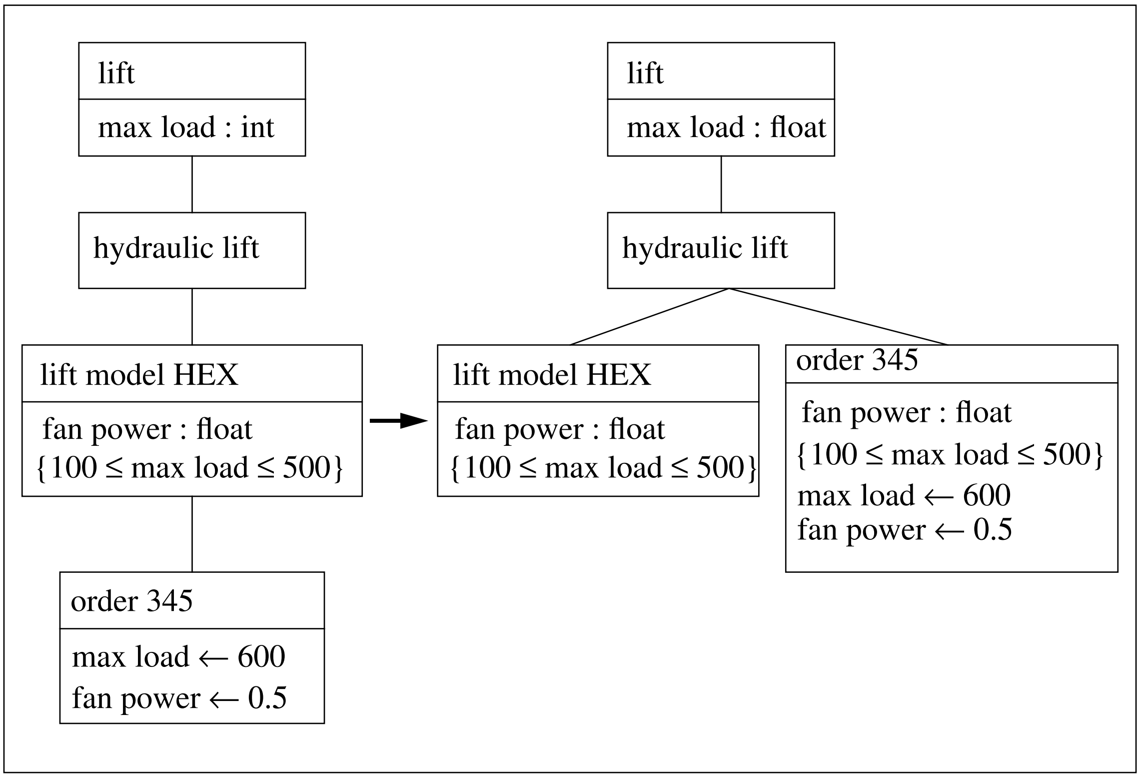


Figure 8: Object generalization

The designer therefore generalizes the order into a hydraulic lift. The order is still invalid but the violated constraint is now local and can be relaxed or deleted in the order. The declaration of *fan power* is automatically copied from *HEX* to *order 345*, which allows the order to preserve the value assigned to the attribute.

In some prototype languages, such as SELF [17], a new object is created by making a copy (*clone*) of an existing prototype object. In our model, this effect is achieved by first creating a new object with the prototype object as the parent and then generalizing the new object to become a child of the parent of the prototype object.

4.6 Composite Objects

The model has been designed to represent physical artifacts. Although on the physical level every component in an artifact is a unique entity, it is not desirable to represent each component (e.g., a simple screw) in the data model as a separate instance. Nevertheless, some components should have a unique identity in the data model as well; this paper concentrates on their representation. The term *component* thus refers specifically to such unique components. (Non-unique components can be represented with simple reference attributes.) The semantics of components are discussed in more detail in [7, 8].

Component Attributes. The (unique) components are represented with attributes of type *component*, which store references to other objects. Object Y is a *direct component* of object X if some component attribute in X refers to Y. We use the term *component* for the transitive closure of the *direct component* relationship.

The set of objects to which a component attribute in an object is allowed to refer can be limited with a constraint. A constraint of the form "$a \le C$", where a is a component attribute and C is an object is true if the value of a is a reference to object C or a descendant of C.

The assignment to a component attribute is treated in a special way. Suppose object X that possesses the declaration of a component attribute should have object Y as a component. When an assignment to the attribute is added to X, the system automatically creates a new instance[3] of Y, i.e., a new object with Y as the parent, and a reference to this new *component object* is assigned to the attribute. The name of the automatically created object is usually unimportant; in this paper the component instances of Y are named as Y-1, Y-2, etc. When an assignment to a component attribute is deleted, the referenced component object is also deleted automatically. For each component object there is thus exactly one attribute assignment with a reference to the object. During the configuration process, however, we allow a limited form of component sharing by means of inheritance. This will be treated in more detail shortly.

Typically the physical components in a product are copies of standard components. An object representing a standard component is thus given as the parent when a component

3. Although we abandon the strict type-instance model, in some cases it still seems appropriate to use the term instance for the children of an object.

is created. If the component has no data which is specific to the particular copy of the standard component, the component need not contain any attribute declarations or assignments of its own.

Figure 9 shows object *lift*, which contains a declaration for component attribute *door*. The constraint in the object means that the object cannot be valid unless the attribute refers to *lift door* or any of its descendants. Object *order 456* has *lift* as its parent and inherits the attribute declaration. Object lift door represents any kind of door and object *XYZ door* represents a particular door type with attribute *width*, that can take values between 1000 and 1500 mm.

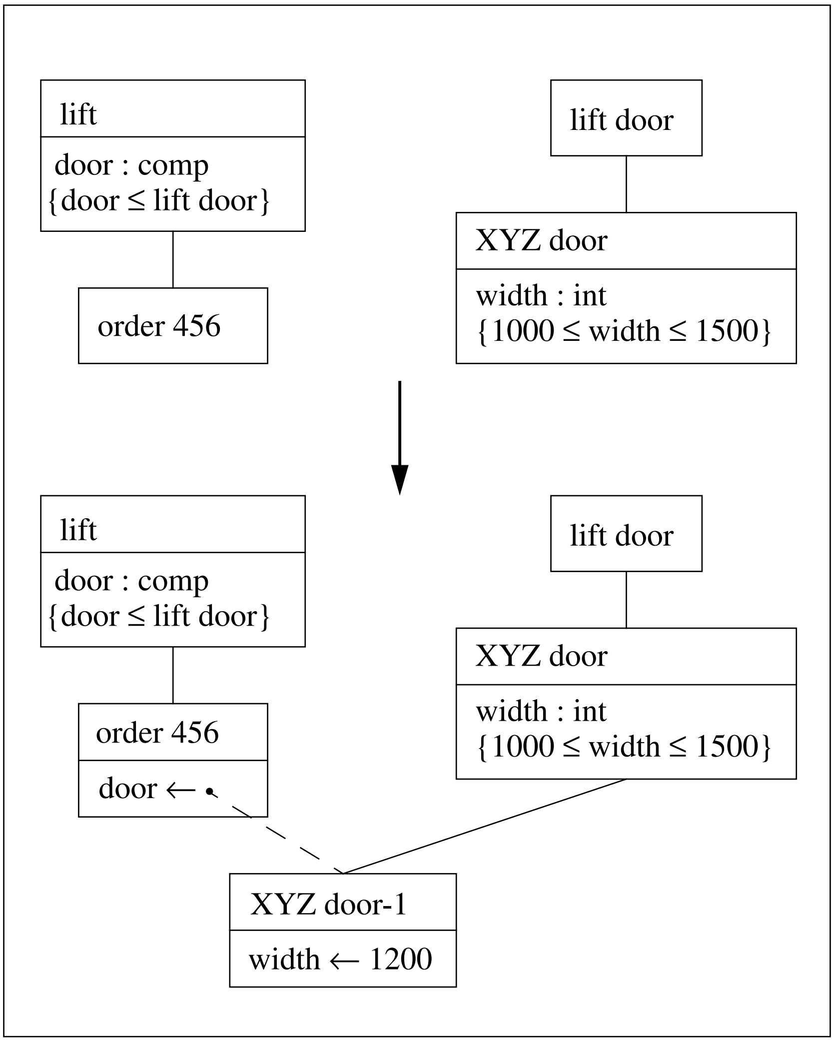


Figure 9: Adding a component

Now suppose *order 456* should have an *XYZ door* as a component. An assignment to attribute door is added to *order 456*. The value is a component reference, marked with a dashed line, to a new component object with *XYZ door* as the parent. An assignment to attribute *width* can then be added to the component object.

Figure 9 shows that lifts in general should have a component of the type *lift door*. One may wonder why this fact is not represented as an assignment together with a corresponding graphical notation, i.e., some kind of a line between the objects *lift* and *lift door*.

We use constraints, such as "$door \leq lift\ door$", because they can express more complex rules. For example, we can say that a lift has either XYZ door or ABC door with the constraint "$door \leq XYZ\ door \vee door \leq ABC\ door$". Similarly, the door can be made optional with the constraint "$door = \text{NULL} \vee door \leq \ldots$".

Nevertheless, constraints of the type "$X \leq Y$" will probably be quite common for component attributes. A graphical user interface for the model should therefore recognize this and other typical constraints as special cases and display them in a more intuitive way.

A component object cannot be used as the parent of any object (however, see component specialization later). Nevertheless, an object with components can freely be used as a parent. Consider the upper part of Fig. 10, which shows object *order 567* with *order 456* of Fig. 9 as the parent. Since component assignments are inherited in the same way as other assignments, attribute door in *order 567* refers to the same component as in *order 456*. Both *order 456* and *order 567* thus have the same object as a component.

Component Copies. Continuing on Fig. 10, suppose the parent of *order 567* is changed to be lift. According to Sect. 4.5, the inherited attribute assignment in *order 456* should be copied to *order 567*. However, since there cannot be two component references to the same object, object *XYZ door-1* is copied and the copied assignment in *order 567* will refer to this copy. More precisely, a new object *XYZ door-2* with *XYZ door-1* as the parent is created,[4] *XYZ door-2* is changed to have the same parent as *XYZ door-1*, and the assignment which is copied to *order 567* is changed to have *XYZ door-2* as attribute value.

Suppose *XYZ door-1* has components. When this object is copied by changing *XYZ door-2* to have the same parent as *XYZ door-1*, the above rules for parent change are applied recursively. As a result, the components of *XYZ door-1* are copied to become components in the copy of *XYZ door-1*.

Shared Components. We do not allow several component attribute assignments to refer to the same object. However, the inheritance of assignments makes it possible for several objects to share component descriptions temporarily during the configuration process.

Suppose the designer is configuring a lift group, which contains two similar lifts and some common control electronics. Both lifts have a car as a component. Eventually there will be two separate physical lifts and two physical cars, each represented with a separate object. While the lift group is being designed, however, the identical properties of the lifts, such as their cars, should be described only once. As shown in Fig. 11, the common properties can be represented with an auxiliary object *lifts of 999*, which serves as the par-

---

4. During this operation we temporarily break the rule that a component cannot be used as a parent.

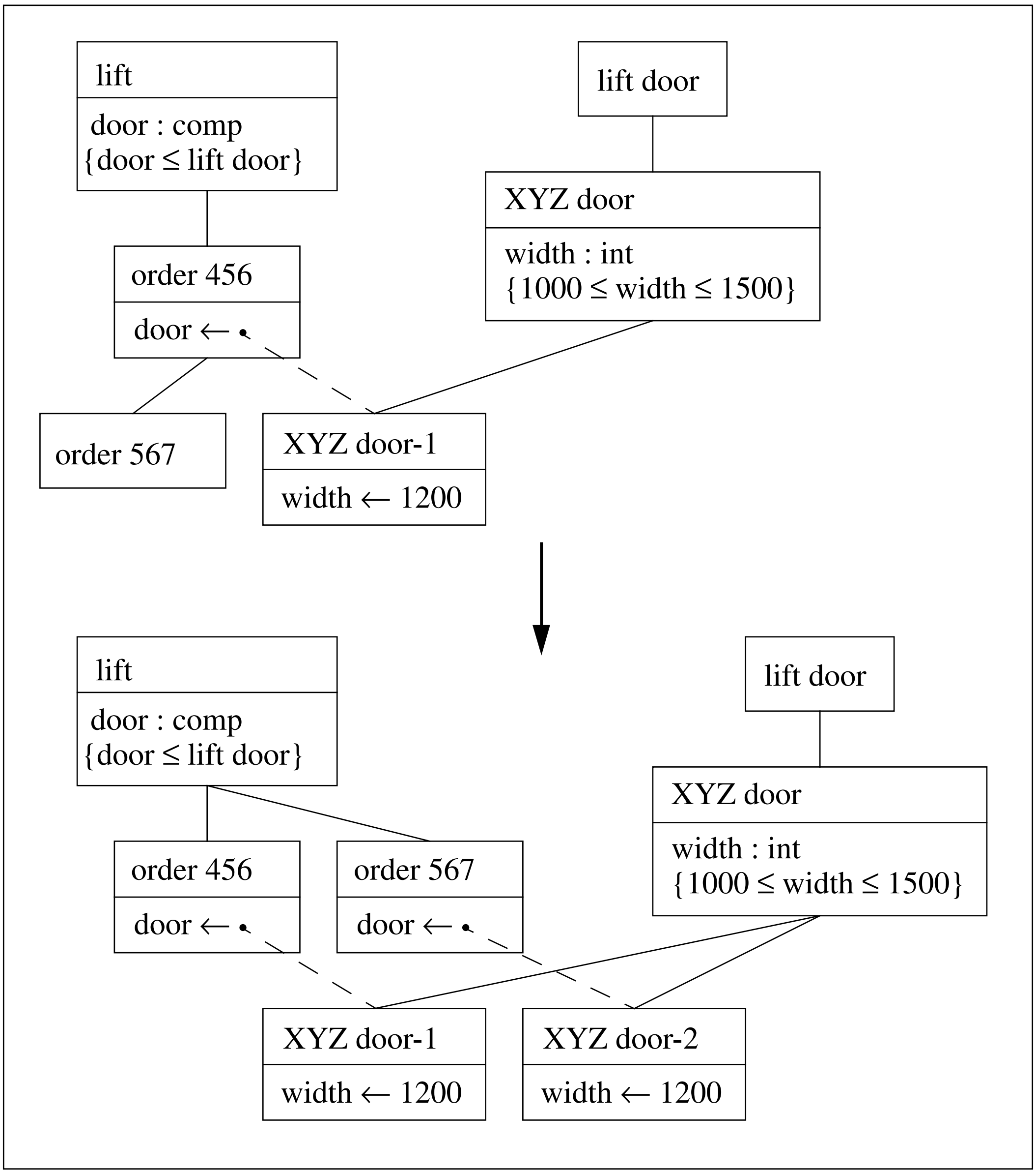


Figure 10: Copying a component

ent for both lifts in the lift group. Both *lift 1* and *lift 2* thus refer to *ABC car-1*, which represents the common properties of the two future physical objects.

However, at some point the two cars are separated in the model. The two objects correspond to two separate physical entities; they have their own identities and their properties can be changed independently.

Instantiation. After the manufacture of the product described by an object has been started, the object no longer represents a plan for the object but the actual physical object. The object must therefore be *instantiated* so that it is not affected by changes in other objects (i.e., its ancestors).

One possibility is to copy all effective inherited assignments to the object. Since the constraints are still inherited from the ancestors, the validity of the instantiated object

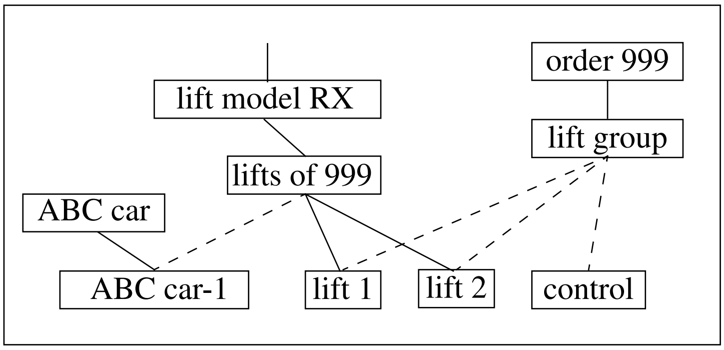


Figure 11: Shared components

depends on the constraints of its ancestors. In other words, the object is still regarded as an instance of the type specified by the parent.

Another way to instantiate an object is to generalize it to have the root object as the parent. Since the root object does not contain any properties, all properties that the instantiated object has inherited from other objects are copied to the object as local properties. An object instantiated in this manner becomes fully "self-contained"; it does not inherit any properties from other objects.

All components of the instantiated object are instantiated recursively in the same way as described above.

Component Specialization. A component object cannot be used as a parent for other objects because it would be difficult to specify the meaning of this construction. (What would actually be the component?) However, it is quite conceivable that sometimes one wants to use an existing component as a basis for developing new components. This situation is handled with a *component specialization* operation. When a component assignment in an object is specialized, a new object with the original component as the parent is created, and the assignment is changed to refer to this new object. The original component can be now used as a parent for other objects.

5 Configuration Process

The object model presented in the previous section is very general. A conventional type hierarchy with instances has a standard interpretation: Each object type represents a set of possible object instances, and each instance corresponds to an entity in the "real world". A specialization hierarchy of the prototype model does not lend itself to such obvious interpretation. In this section we describe how the prototype model can be used for modelling a configuration process which is carried out in multiple phases.

Consider a lift model, such as the RX lift model of Fig. 3. A configuration of this lift could for instance be created in the following steps:

- A specification is created. The lift model has constraints for checking the validity of the specification. For example, the lift model can have a constraint for the possible speed range, and the specification assigns a value to the corresponding attribute.[5]
- After the specification has been checked, the first phase of configuration process is carried out. The lift model specifies attributes that must be assigned during this phase and constraints that the configuration must satisfy at the end of the phase. The decisions of the first phase represent a certain commitment by the manufacturer. It is, for example, possible that the price of the product is fixed during this phase or the manufacture of some components is started.
- When the configuration satisfies the constraints of the first phase, the second phase is performed. New attributes and constraints are introduced.
- When the constraints of the second phase are satisfied, the product represented with the configuration can be manufactured.

Figure 12 shows these steps using the object model. Object *lift model RX* contains the attributes and constraints for the specification. The processing of a new *order 678* begins with the creation of an object with *lift model RX* as the parent (situation 1). After this object has been made valid, i.e., the specification constraints are satisfied, one can create a new object for phase 1 data with the specification as the parent. The specification is then changed to have phase 1 description as its parent (situation 2). The object for phase 1 data now inherits new attributes and constraints from phase 1 description. After phase 1 data is valid, phase 2 is carried out in a similar way (situation 3).

Note that it is not necessary to represent the specification, phase 1 data and phase 2 data of the order as three separate objects. We could have a single object for the order. During the configuration process the parent of this object would be changed as described above, but the attribute assignments of each phase would be added to the single object. The separate objects, however, make it possible to record what data was added to the configuration during each phase. Typically these objects also store process related data, such as when the data was modified and by whom.

Situation 3 in Fig. 12 still looks somehow strange. For example, phase 1 data is a descendant of phase 2 description. This means that the object for phase 1 data is probably invalid because it only contains data that was added during phase 1 although it now inherits additional constraints from phase 2 description. One can sensibly examine only the validity of the data for the "current phase". In situation 3, for example, one can only ask whether the object for phase 2 data is valid.

After a configuration is ready and the corresponding lift has been manufactured and delivered to the customer, the configuration is moved to a new place within the object tree. Figure 13 shows how the configurations of the delivered lifts are stored as ancestors of a single object, which can declare attributes associated with the delivered lifts, such as the

5. Note that we can only check assigned values against constraints. In general it is impossible to check for inconsistencies between constraints.

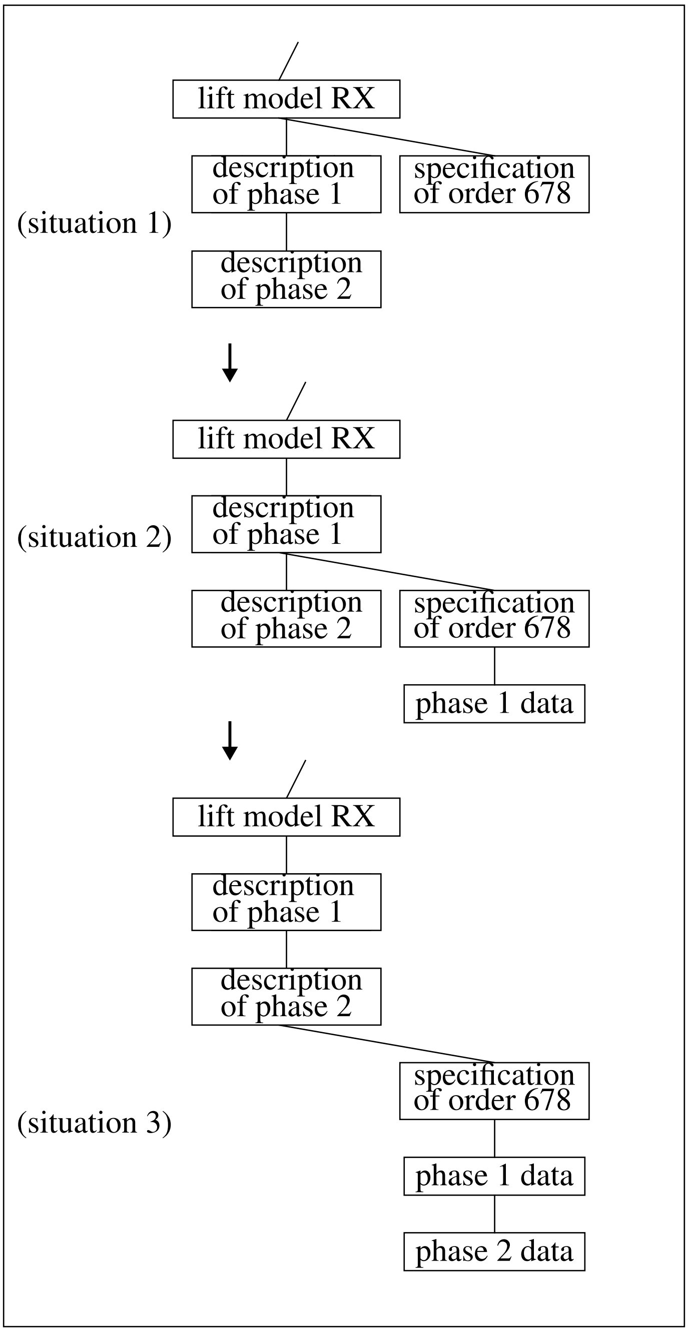


Figure 12: Configuration process

delivery date. Many of these attributes are actually common to deliveries of all lift models; multiple inheritance is therefore again likely to be used.

6 Conclusions and Future Work

We believe that product configuration management is a good example of an application which is much better served by the prototype object approach than the more common

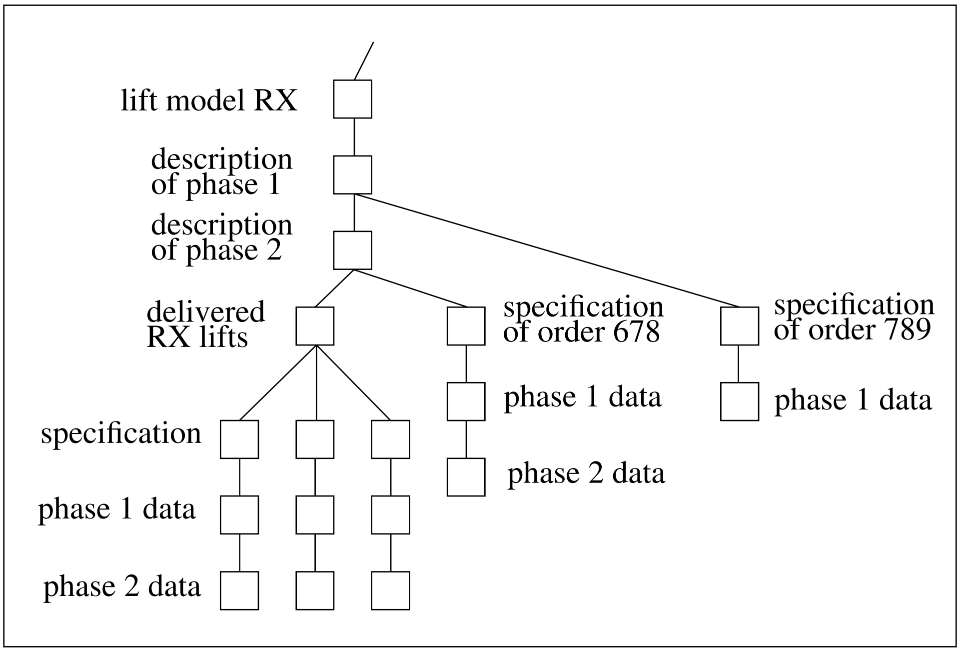


Figure 13: Delivered and in-progress configurations

type-instance model. This property seems to stem from the inherent dynamism of the configuration problem. Operations that are usually classified as schema updates performed by a database administration are part of the routine data manipulation during the configuration process.

The generality of the prototype model means that an application must adopt particular conventions for using the objects. As explained in Sect. 5, we organize the objects in a hierarchy which represents the generic properties of lifts, the standard products, and the individual configurations in various phases of the configuration process.

The limited space has prevented us from touching many important issues of the model. These topics, which we hope to be able to treat in future papers, include:

- *Multiple inheritance*. As seen from the examples, the model requires multiple inheritance for representing various situations encountered in actual configuration process. One such situation is the problem discussed in Sect. 5 in connection with Fig. 12. As illustrated in Fig. 14, relationships between the phases can be expressed better when phase 2 data has both phase 1 data and phase 2 description as its parents. Nevertheless, multiple inheritance complicates the model considerably.

- *Versioning*. The uniform treatment of objects means that a single mechanism can be used for "type versions", i.e., for schema evolution, and for more conventional "instance versions". The tree transformations of Sect. 4.5 are important for type evolution.

- *Configuration processes*. This paper is mainly concerned with the representation of configuration data. We have not discussed how designers select proper components to configurations and how they determine proper values for attribute assignments. For this purpose, we plan to include selection and computation procedures in the objects. We also need a mechanism to describe configuration processes, such as the phases outlined in Sect. 5.

- *Environments*. According to this paper, all objects reside in single large tree. We are developing a mechanism for extracting smaller parts of the tree to separate environments. This mechanism will be similar to some concepts of Sun Network Software Environment (NSE) [14].
- *System architecture*. The configuration system will be connected to other systems, such as CAD programs, document management systems and programs for computing attribute values.

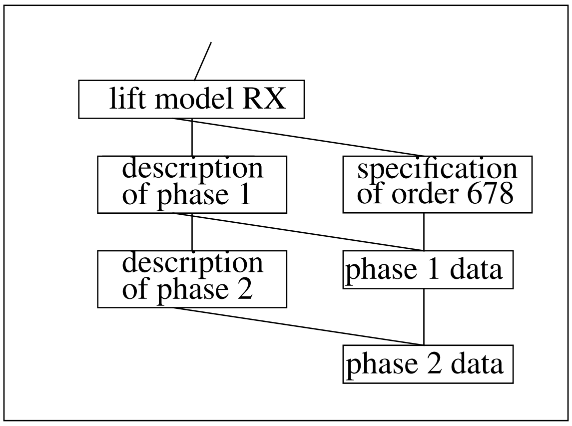


Figure 14: Configuration phases with multiple inheritance

Acknowledgments

This work is carried out in a joint project with KONE Elevators. The project is partly funded by the Finnish Technology Development Centre (TEKES) as part of the SIMSON programme of the Federation of the Finnish Metal, Engineering and Electrotechnical Industry (FINMET). The SIMSON programme is also a partner in the IMS/GNOSIS programme.

References

[1] Jay Banerjee, Won Kim, Hyoung-Joo Kim and Henry F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. International Conference on the Management of Data (SIGMOD)*, pages 311–322. 1987.

[2] Frans Coenen and Trevor Bench-Capon. *Maintenance of Knowledge-Based Systems*. Academic Press 1993.

[3] Adrian Demaid and John Zucker. Prototype-Oriented Representation of Engineering Design Knowledge. *Artificial Intelligence in Engineering*, vol 7, pages 47–61, 1992.

[4] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[5] ISO Standard 10303-11 Industrial Automation and Integration—Product Data Representation and Exchange—Part 11: Description Methods: The EXPRESS Language Reference Manual.

[6] Tsuneyoshi Katsuama, Hirokazu Taki, Hidekazu Tsuji, Akihito Naito, Motonori Yoshida and Kihatirou Ohnishi. An Expert System for Elevator Design. In *Proc. of the World Congress on Expert Systems 1991*, pages 36–45, 1991.

[7] Won Kim, Jay Banerjee and Hong-Tai Chou. Composite Object Support in an Object-Oriented Database System. In *Proc. International Conference on the Management of Data (SIGMOD)*, pages 118–125, 1987.

[8] Won Kim, Elisa Bertino and Jorge F. Garza. Composite Objects Revisited. In *Proc. of the International Conference on the Management of Data (SIGMOD)*, pages 337–347, 1989.

[9] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 214–223, 1986.

[10] Sandra Marcus, Jeffrey Stout and John McDermott. VT: An expert elevator design that uses knowledge-based backtracking. In *Artificial Intelligence in Engineering Design. Volume I*, chapter 11, pages 317–355. Academic Press Inc., 1992.

[11] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine*, vol. 6, no. 4, pages 40–62, Winter 1986.

[12] Lynn Andrea Stein. Delegation Is Inheritance. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 138–146, 1987.

[13] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–93, 1991.

[14] Sun Microsystems Inc. *Introduction to NSE*, 1988.

[15] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD Thesis, University of Jyväskylä, Finland, 1993.

[16] Chris Tong and Duvvuru Sriram. Introduction. In *Artificial Intelligence in Engineering Design. Volume I,* chapter 1, pages 1–53. Academic Press Inc., 1992.

[17] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. *LISP and Symbolic Computation,* 4(3), 1991.

[18] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger,* 1(1):7–87, August 1990.

[19] Gregg R. Yost. Configuring elevator systems. Technical report, Digital Equipment Corporation, 1992.

[20] Roberto Zicari. A Framework for Schema Updates in an Object-Oriented Database System. Chapter 7 in F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System—the Story of O_2*. Morgan Kaufmann Publishers, 1992.