An Object Model for Evolutionary Configuration Management

Hannu Peltonen, Tomi Männistö, Reijo Sulonen and Kari Alho
Helsinki University of Technology
Department of Computer Science
Otakaari 1, 02150 Espoo, Finland

Abstract

An object model for evolving engineering design data is presented. The model is based on prototype objects and includes tree transformations for object generalization and specialization during a design process. In addition to attribute data, objects contain constraints for checking their validity. Objects are arranged into component hierarchies, and constraints can express dependencies between arbitrary objects in such hierarchies.

1 Introduction

In many manufacturing industries products are built from predefined components according to customer specifications. In order to fulfil the specification, the designer must select appropriate components and determine suitable values for various parameters associated with the components.

Often the designer has a range of standard products available as a basis for his or her work. A standard product defines some components and provides default values for parameters, reducing the effort to create a detailed description of the delivered product. We call the task of creating a product description from components a *configuration process.* The process is illustrated in Figure 1.

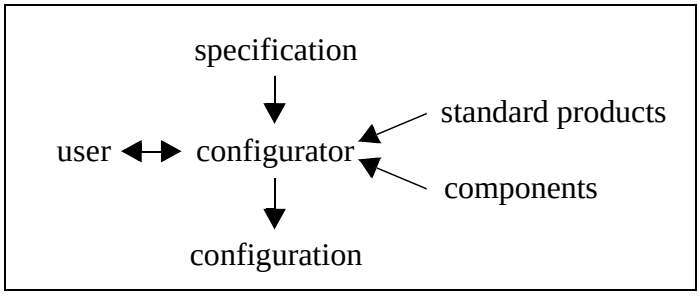


Figure 1: Configuration process

We want to study object models for representing the configuration process. From our point of view the problem has two important aspects we want to emphasize: gradual refinement and graceful evolution. The configuration process—like design processes in general—is an

iterative, sometimes experimental process where many different representations, some of them quite tentative, may have to be created. Designs and their representations, in terms of design objects, or simply objects, grow and evolve during the design process. Properties of objects may be added, modified, or dropped quite freely; both non-structural, e.g., changing attribute values, and structural changes, e.g., adding new attributes or methods, have to be accommodated. The design proceeds from abstract to concrete by gradually incorporating more details through design decisions, or from potentiality to actuality using the terms of Aristotle. In principle, a finished design represents an unambiguous, "real" product, while an unfinished design still stands for a whole set of acceptable products. Thus, to us it would be an advantage to have a framework which could nicely support some form of gradual refinement of designs.

The life-times of many industrial products are quite long, often tens of years; in construction industry even much longer. During the life-time of a product two different forces drive the changes. On one hand the delivered products are being maintained or enhanced on an individual basis; for instance, two similar products delivered to different customers may end up quite soon being quite different. On the other hand, the design knowledge, i.e. the components and standard products, used at any particularly moment to configure new products is evolving, in principle, independently from changes to previously delivered products. We may run into a situation where we want to replace a component of a product delivered five years ago by a new component using the validity rules of the original product, possibly long ago rejected from the product line. We call catering for these two "orthogonal" change processes as *graceful evolution.*

The prototype approach [6, 8, 9] seems to provide us with the flexibility needed to cope with gradual refinement and graceful evolution. The fundamental idea is to allow each object to serve both as a representation of some "rudimentary version" [3] and as a source for further refinements; there is no distinction between classes and instances. We have elaborated the prototype approach using single inheritance for attributes, their value assignments and constraints. In the model every object may serve as a type for its children which are instances of their parent; however, types and instances are strictly in the eyes of the viewer. The constraints are used to express conditions valid objects and their valid descendants have to satisfy. With respect to validity, an object can be mutated as long as it remains within its

constraints. To support more radical object changes, we have introduced a number of tree transformation operations akin to specializations and generalizations. Special semantics is associated with composite structures to represent the familiar hierarchical product structure; global constraints can be specified between components within the product structure.

The rest of the paper is organized as follows. Sections 2 deals with objects, attributes and constraints. Section 3 introduces composite objects, which are used to represent configurations. Section 4 describes more complex constraints, which express the conditions for valid configurations. The constraints allow a designer to check the validity of a configuration (both internally and with respect to the specification). Finally, Section 5 provides a brief summary and discusses further work.

2 Basic Object Model

All components, standard products and configurations are represented as objects. An object is a collection of *object elements*, which comprise *attribute declarations* (Section 2.2), *attribute assignments* (Section 2.3) and *constraints* (Section 2.4). The idea of an object type (or class) has been rejected; nevertheless an object can inherit elements from other objects.

The model specifies operations for manipulating the attributes and constraints of the objects. At this point, the objects do not contain methods or other means to describe user-defined behavior. The objects only store attribute values and constraints for checking the validity of the objects.

2.1 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. Section 2.5 explains how an object can be changed to have a new parent. The root object may not contain any elements.

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

Each objects has its own identity and a name for reference. The detailed format of object names is irrelevant in this paper.

2.2 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 Section 3.

A newly created object does not contain any elements. An attribute declaration can be added to any object (except the root object) provided that the object does not already contain a declaration for an attribute with the same name.

However, an object and some of its ancestors can declare attributes with the same name. This paper will occasionally mention the possibility of this kind of *name conflict*, but its full treatment is beyond the scope of this paper.

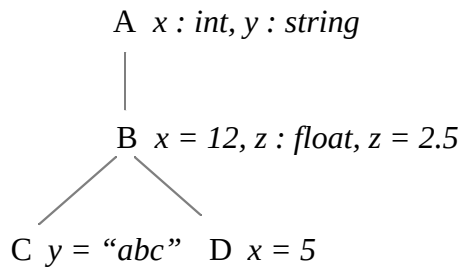
2.3 Attribute Assignments

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[1] of an attribute and a value, which must conform to the attribute type in the declaration. The value in an assignment can be later changed.

The value of a particular attribute in a particular object is read from the first assignment to the attribute on the path of objects from the given object towards the root object along the parent links. If the path does not contain any object with an assignment to the attribute, but 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.

Figure 2 shows four objects with attribute declarations and assignments. The dotted 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.

1. The connection between attribute declarations and assignments is complicated by the possibility of name conflicts. The details are not presented here.



Object	Attribute values		
	x	y	z
A	?	?	error
B	12	?	2.5
C	12	"abc"	2.5
D	5	?	2.5

Figure 2: 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 because the assignment is inherited, but any descendant can add an assignment to the same attribute.

2.4 Constraints

In addition to attribute declarations and assignments, an object can possess constraints, which specify the conditions for the validity of the object. Constraints make it possible to check the validity of a configuration since configurations will be represented as composite objects (Section 3), and constraints can be defined for composite objects (Section 4). The representation of the specification of a configuration by means of constraints will be discussed in Section 4.5.

A constraint is an expression which evaluates to *true*, *false*, or *unknown*, and is composed of literals, references to attributes and usual numeric and string operators.

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. A more precise definition for the validity of composite objects will be given in Section 4.

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

At the moment, the model only defines how the validity of an object is checked but says nothing about when this is done or how the system reacts to an invalid object. In any case, the system can contain invalid objects, which often arise during a configuration process. These issues are connected with the concept of a *design transaction* [1].

2.5 Parent Change

As was explained in Section 2.1, 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 object represents a subset of the possible objects of its parent object.

2.5.1 Specialization

When the new parent of an object is a descendant of the old parent, the object is "specialized". It may inherit more attributes and constraints than before and represents a subset of possible objects.

In Figure 3, an elevator order is represented with the object *order-123*. Originally the order is only specified to be some kind of a hydraulic elevator. Later during the configuration process, the designer selects HEX from the available standard elevators as a basis for the order. This standard elevator includes an automatic ventilation system and accordingly object HEX contains a declaration for attribute *fan_power.* After *order-123* is changed to have HEX as the parent, an assignment to *fan_power* can be added to *order-123*.

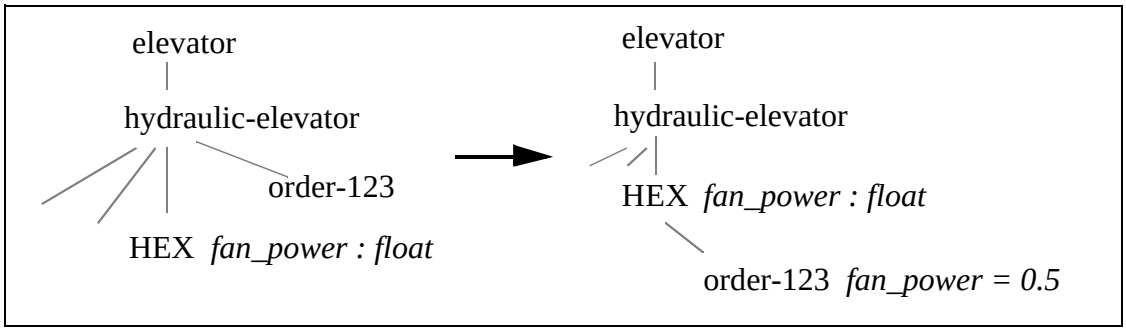


Figure 3: Object specialization

2.5.2 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 may inherit fewer elements 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.

To preserve some of the semantics of the object, we copy the intervening elements (declarations, assignments and constraints) to the object. The generalization of an object preserves the elements the object possesses. However, some attributes and constraints become "local properties" that can be modified without affecting other objects (except of course the descendants of the modified object). This approach differs from [2] 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 [10].

Figure 4 shows the same objects as Figure 3. Elevator has an attribute *max_load*, which is constrained in the standard elevator HEX to lie between 100 and 500. The maximum load of *order-123* has been assigned value 600. Since this violates the inherited constraint, the order no longer belongs to the set of valid elevators as specified by the standard elevator HEX.

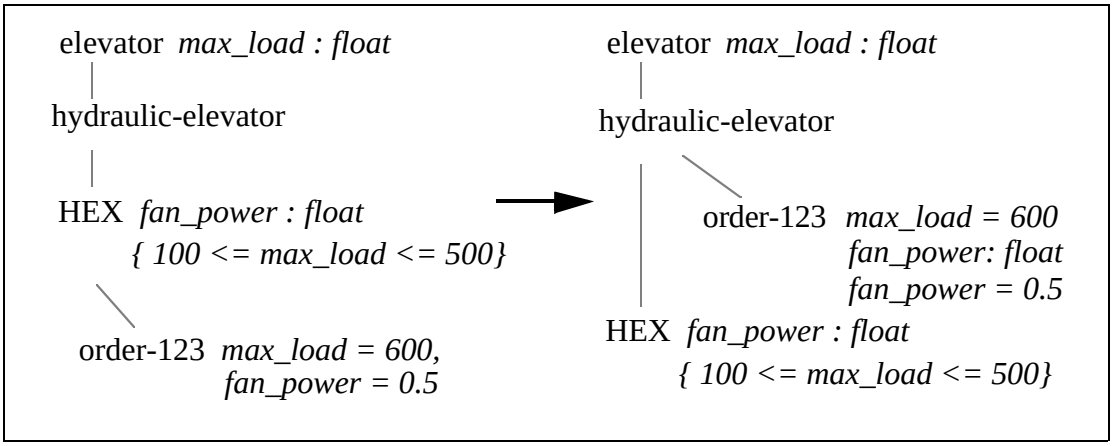


Figure 4: Object generalization

The designer therefore generalizes the order into a hydraulic elevator. The order is still invalid but the violated constraint is now local and can be relaxed or deleted in the order (it has been deleted in Figure 4). The declaration of *fan_power* is automatically copied from HEX to *order-123*, which allows the order to preserve the value assigned to the attribute.

2.5.3 Combined Parent Change

It is also possible that the new parent is neither a descendant nor an ancestor of the original parent. In this case, the "lowest common ancestor" of the original parent and new parent is found. The object is first generalized "up" to the common ancestor and then specialized "down" to the new parent.

2.6 Types and Instances

The model does not incorporate concepts of an object type and instance. In some sense, one can regard an object with children as a type and its children as instances of that type. On the other hand, the children can often also be seen as subtypes. The distinction between an instance and a subtype is never absolute in our model because any instance can serve as a parent for another object and thus become a subtype.

Our rationale for dispensing with types and instances in the description of engineering data is very similar to that of Demaid and Zucker [3]. Our model also resembles the hybrid model in [9].

If parent objects are regarded as types, it is also possible to talk about "type evolution" [2, 7] in our model. Object T in Figure 5 represents a type, which declares attributes *a* and *b*, while object X represents an instance of this type. Now we want to create a new version of type T, called T2, in which attribute *b* is deleted and a new attribute *c* is added. Then we also want to change instance X to be of this new type.

First we create object T2 with T as its parent (situation 2). Then we change the parent of T2 to be S (situation 3); this copies all declarations of T to T2. Then the declaration of *b* is deleted from T2 and the declaration of *c* is added (situation 4). Next we want to "coerce" X to conform with type T2 by changing its parent to be T2 (situation 5); the declarations in T are copied to X.

X no longer inherits attribute *b* from its type. Nevertheless, the attribute is still available in X as a local declaration. X inherits the new attribute *c* from the new type. Attribute *a* becomes ambiguous in X. Typically the declaration of attribute *a* in X is now removed and all references to the attribute are replaced with references to the attribute *a* in T2 (situation 6b). It is, however, possible that attributes with the same name in the old and new type version do not

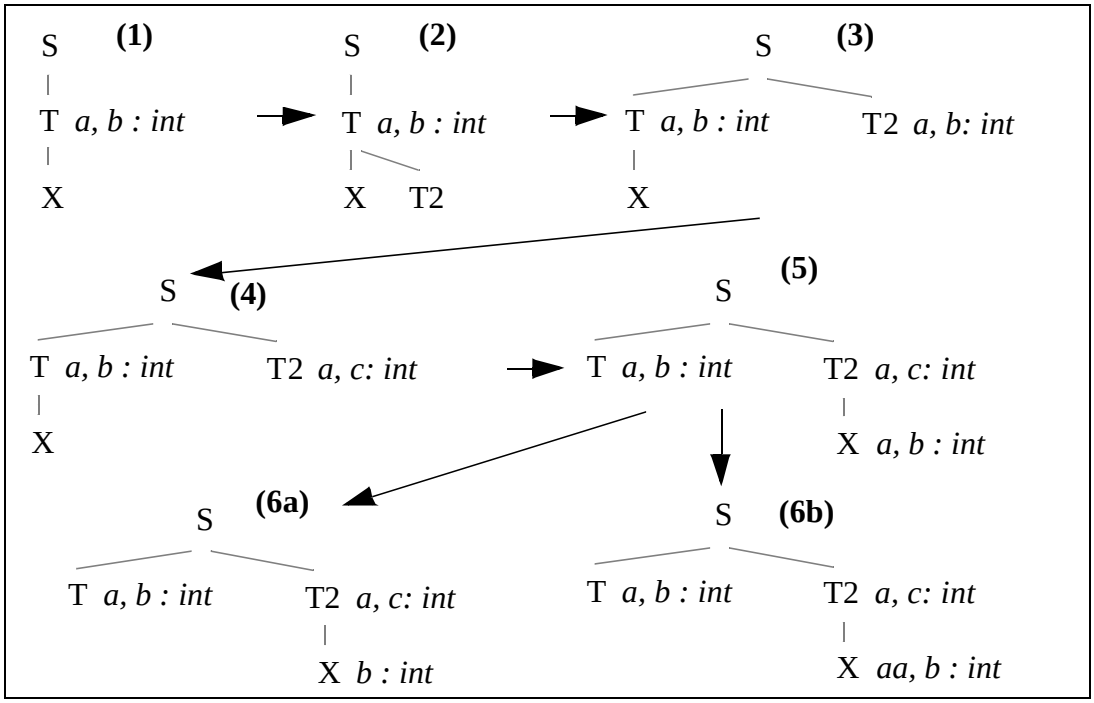


Figure 5: Type evolution

represent the same data. If X must have both the "old" and "new" attribute, attribute *a* in X is renamed (situation 6b).

Note that we could make a copy of the instance X in exactly the same way. First we create a new object X' with X as the parent, and then we generalize X' to have T as the parent.

Of course, this mechanism does not solve the type evolution problem. It, however, gives us the handle to deal with the problems in a meaningful way. We are developing the model mainly as a data representation for configuration processes. Therefore, unlike Skarra and Zdonik [7], we have not addressed the problem of application programs that must access instances of one version of a type as if they were instances of another version of the type.

3 Composite Objects

Configurations are represented as composite objects, which have other objects as components. We make a distinction between simple *references* and actual *component links*, which have special semantics and can be associated with special constraints for describing valid configurations [4, 5].

3.1 Reference Attributes

The value of an attribute of the type *reference* is a reference to some object. If it is necessary to limit the set of objects that a reference attribute in an object can take as a value, a constraint can be added to the object. A constraint of the form "a ≤ C", where *a* is a reference attribute and C is an object is true if the value of *a* is a reference to object C or a descendant of C. These constraints can also be used with component attributes (see below).

3.2 Component Attributes

An object can have other objects as *components*. The components are represented with attributes of type *component*, which, like reference attributes, have references to other objects as values. 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.

If a composite object is considered to represent a physical part hierarchy, the components should always form a strict tree because a single part cannot be physically included in more than one object [5]. For each object X, there can be at most one assignment to a component attribute with X as the attribute value. Whenever an assignment to a component attribute is added to an object, a new object is created and a reference to this object is used as the attribute value. During the configuration process, however, we allow a limited form of component sharing by means of inheritance. This will be treated in more detail in Section 3.4.

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 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 6 shows object *elevator*, which contains a declaration for component attribute *door*. Component relationships are marked with solid lines. The constraint in the elevator object forces the attribute to refer to *elevator_door* or any of its descendants. Object *order-123* has *elevator* as its parent and inherits the attribute declaration. Object *elevator_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.

Now suppose *order-123* should have an *XYZ door* as a component. An assignment to attribute *door* is added to *order-123*. The value is a reference to a new object with *XYZ door* as the parent. An assignment to attribute *width* can then be added to the component object.

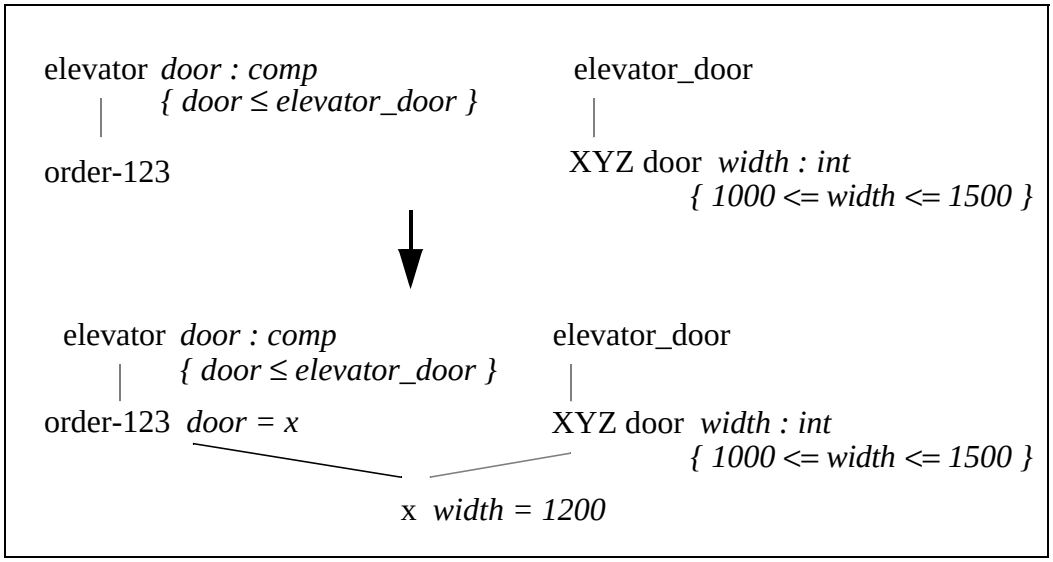


Figure 6: Adding a component

A component object cannot be used as the parent of any object (however, see Section 3.5). Nevertheless, an object with components can freely be used as a parent. Consider the upper part of Figure 7, which shows object *order-456* with *order-123* of Figure 6 as the parent. Since component assignments are inherited in the same way as other assignments, attribute *door* in *order-456* refers to the same component as in *order-123*. Both *order-123* and *order-456* thus have the same object as a component (see also Section 3.4).

3.3 Component Copies

Continuing on Figure 7, suppose the parent of *order-456* is changed to be *elevator*. According to Section 2.5, the inherited attribute assignment in *order-123* should be copied to *order-456*. However, since there cannot be two references to the same component, object *x* is copied and the copied assignment in *order-456* will refer to this copy. More precisely, a new object *x'* with *x* as the parent is created[1], *x'* is changed to have the same parent as *x*, and the assignment which is copied to *order-456* is changed to have *x'* as attribute value.

Suppose object *x* has components. When *x* is copied by changing *x'* to have the same parent as *x*, the above rules for parent change are applied recursively. As a result, the components of

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

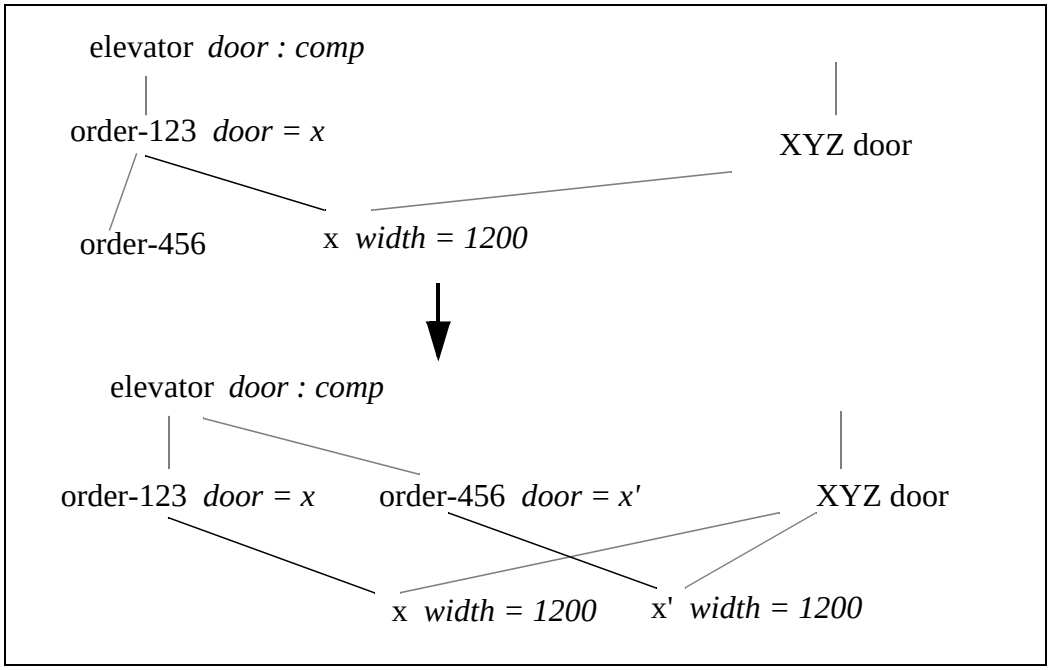


Figure 7: Copying a component

x are copied to become components in the copy of *x*. Note that this rule of "copying the components" only affects component assignments. If object *x* has an inherited component, the copy of *x* simply inherits the same component.

3.4 Shared Components and Instantiation

As explained in Section 3.2, our model does not allow several component attribute assignment to refer to the same object. However, the inheritance of assignments makes it possible for several objects to have common components.

Suppose the designer is configuring an elevator group, which contains two identical elevators and some common control electronics. While the elevator group is being designed, the identical properties of the elevators, such as their cars, should be described only once. As shown in Figure 8, the common properties can be represented with an auxiliary object, which servers as the parent for both elevators in the elevator group.

Object *elevator_group* has two components: *elevator_1* and *elevator_2*, which have *model_elevator* as the parent. Since *model_elevator* has *car_x* as a component, *elevator_1* and *elevator_2* also have this component. All changes in *car_x* thus affect both elevators. However, when the elevator group is actually manufactured and installed, the two cars should be *instantiated*, i.e., they should have their own identities because they now correspond to two

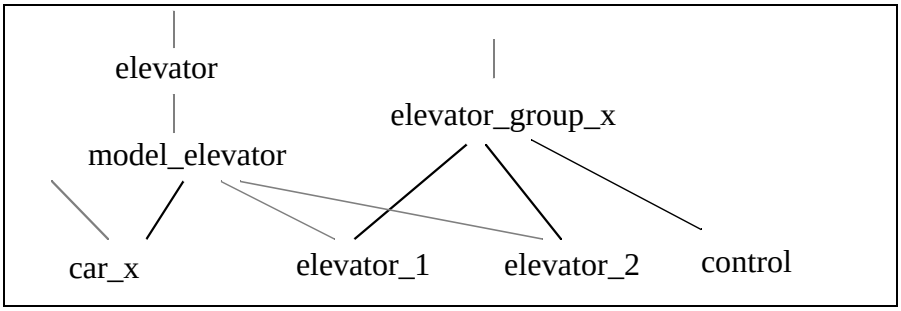


Figure 8: Shared components

separate physical entities. For example, components in one car can be changed without affecting the other one.

One way to instantiate an object is to change it to have the root object as the parent. Since the root object does not contain any elements, all elements that the instantiated object has inherited from other objects are copied to the object as "local" elements. When *elevator_1* and *elevator_2* in the above example are changed to have the root object as the parent, they will, among other things, have their own copies of *car_x*. An object instantiated in this manner becomes fully "self-contained"; it does not inherit any elements from other objects.

An alternative is to leave the object with its original parent, but to copy all inherited assignments. (Individual assignments can be copied from the parent object; components are copied recursively in the same way as when the parent is changed.) The object is then not affected by any changes in the attribute values—including component assignments—of the parent object. However, since the constraints are still inherited from the parent object, the instantiated object must satisfy the constraints of the parent. In other words, the object remains an instance of the type specified by the parent.

3.5 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.

4 Constraints for Composite Objects

4.1 References to Other Objects

A constraint in an object can refer to other objects by means of reference and component attributes. For example, suppose an elevator includes a *car* and a *motor.* The car has attribute *weight* and the motor has attribute *max_load*. The weight of the car must not exceed the maximum load of the motor.

If the elevator has component attributes *car* and *motor*, the following constraint can simply be added to the elevator: “car.weight <= motor.max_load”.

Typically the constraint would be part of a generic description of an elevator and the constraint would be inherited by all specific elevators that have the general elevator object as their ancestor.

4.2 Component DAGs and Roles

Suppose all elevators have a car and motor, but they can be located at different levels of the component structure in different elevators. An elevator could, for example, have a component *driving mechanism*, which has the motor as a component. What is needed is a mechanism by which the elevator can refer to motor component regardless of its exact location in the component structure. One possible solution is outlined below.

An object can be assigned an arbitrary string as a *role*. For example, the elevator group in the earlier Figure 8 has the components shown in Figure 9. The role of each object is shown in brackets after the object name.

A single object can be a component of several objects as a result of inheritance of component assignments. The components of any given object thus form a DAG with one or

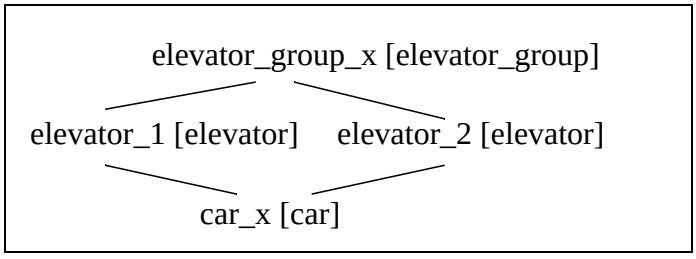


Figure 9: Component DAG

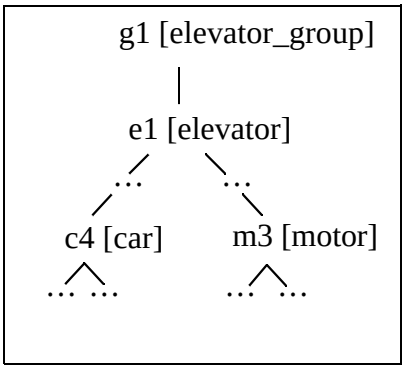


Figure 10: Elevator Components

more paths from the object to each of its components. For example, component *car_x* of *elevator_group_x* in Figure 9 is reached along paths *elevator_group_x/elevator_1/car_x* and *elevator_group_x/elevator_2/car_x*. The *role paths* of the components of an object are formed by replacing the objects in the component paths with their object roles.

A constraint in an object can refer to components of the object by role names using function

comps(*role path pattern*)

The argument is a role path where some roles may have been replaced with an asterisk. The value of the function in object X is a set of objects. The set includes those components of X that have a role path within X that matches the function argument. An asterisk in the argument matches zero or more consecutive roles in the role path of a component. A component is included only once in the result even if the component is reached along several paths that match the pattern.

We can now return to the constraint between cars and motors. Suppose an elevator group includes the components shown in Figure 10. The elevator *e1* has components—not necessarily direct ones—*c4* and *m3* in roles *car* and *motor.* Within *g1*, objects *c4* and *m3* have role paths "elevator_group/elevator/…/car" and "elevator_group/elevator/…/motor", respectively.

The elevator must have exactly one component in the role of a *car* and exactly one component in the role of a *motor* and attribute *weight* of the car must not exceed attribute *max_load* of the motor. This can be expressed by adding the following constraint to some ancestor of *e1*:

```
let c = comps("*/car"), m = comps("*/motor")
in size(c) == 1 and size(m) == 1 and
  c[1].weight <= m[1].max_load
```

4.3 Global Constraints

The constraint between elevator car and motor belongs to the description of an elevator as a whole. Nevertheless, sometimes dependencies between components are recorded with one of the components. As an example, suppose there is luxury elevator car, which can also be used in non-luxury elevators. In this case, however, the elevator must have a special type of motor.

This particular car will have the following constraint:

> When a car of this type is used as a component in an elevator, the elevator must have exactly one motor, which is of the type *special_motor.*

This condition can be recorded as the following *global constraint* in the *luxury_car* object:

```
elevator:
  let m = comps("*/motor")
  in size(m) == 1 and m[1] ≤ special_motor
```

A global constraint is like an ordinary constraint except that it has a role name as a prefix. A global constraint of an object can be checked only when the object is considered as a component of another object. Suppose object *c4* in Figure 10 is a descendant of *luxury_car.* The above constraint of *c4* as a component of *g1* is checked as follows:

(1) Object *c4* has role path "elevator_group/elevator/…/car". The role prefix "elevator" is found in the path and the corresponding object is *e1*.

(2) The constraint is checked so that the comps function is evaluated with respect to object *e1*. The function "comps("*/motor")" thus finds object *m3.*

If the prefix were not found in the role path, the constraint would automatically be true. (The constraint says something about the car if it is regarded as a component in an elevator, otherwise the constraint can be ignored by making it true.)

The global rules must have the prefix which specifies how far "upwards" in the component structure one should travel to check the constraint. For example, we only want to find the motor of the same elevator, not the motors of all elevators in the elevator group.

4.4 Validity Rules for Composite Objects

A composite object is *valid* if the global and non-global constraints of the object and its components are true. If a component is reached along several paths, the non-global constraints need only be checked once for the component, but the global constraints must be checked separately for each path.

For example, if one asks whether the car of an elevator is valid, the global constraint between the car and the motor is not checked. The constraint is checked only when the validity of the elevator is examined. It is thus possible that the car is valid although the elevator is invalid because a constraint in the car is unsatisfied.

Also, a global constraint in *car_x* of Figure 9 can be true for one path while the same constraint is false for another path because the function "comps("elevator")" refers to different object depending on the path along which the car is reached.

4.5 Specification

We are now able to represent the specification of a configuration (see Figure 1) as a set of constraints attached to the root component of the configuration. However, new constraints may be added to this object during the design. To make the specification more explicit, it can be represented as a separate object, which only contains the specification constraints. The actual configuration will then be created as a child to this object. This arrangement also makes it easy to create alternative configurations for a single specification.

5 Discussion

We have presented a model for the evolutionary aspects of engineering data, especially in the context of a configuration problem. The model is based on the well-known prototype approach and gains additional power from the inheritance tree transformations. We believe these operations capture important semantics of the configuration process. Object specialization supports gradual refinement of the design. Object generalization allows the designer to say that a design should no longer be restricted by the constraints of a particular standard product. Global constraints allow dependencies between components in an arbitrary component hierarchy to be recorded in the components.

The model must be developed further and evaluated with real-life situations and problems.

We plan to implement a prototype system on the basis of the ideas put forward in this paper.

A crucial element still missing from the model is the mechanism for specifying configuration rules. We shall maintain a distinction between constraints, that tell whether a configuration is valid, and rules, that help designers to create valid configurations.

References

[1] Bancilhon, F., W. Kim, and H. F. Korth. "A Model of CAD Transactions". In *Proc. of the 11th International Conference on Very Large Databases (VLDB)*, pages 25–33. 1985.

[2] Banerjee, J., W. Kim, H.-J. Kim, and H. 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.

[3] Demaid A. and J. Zucker. "Prototype-Oriented Representation of Engineering Design Knowledge". *Artificial Intelligence in Engineering*. Vol. 7, pages 47–61. 1992.

[4] Kim, W., J. Banerjee, H.-T. Chou. "Composite Object Support in an Object-Oriented Database System". In *Proc. International Conference on the Management of Data (SIGMOD)*, pages 118–125. 1987.

[5] Kim, W., E. Bertino and J. F. Garza. "Composite Objects Revisited". In *Proc. International Conference on the Management of Data (SIGMOD)*, pages 337–347. 1989.

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

[7] Skarra, A. H., and S. B. Zdonik. "Type Evolution in an Object-Oriented Database". In B. Shiver and P. Wegner, editors, *Directions in Object-Oriented Programming*, pages 393–415. MIT Press 1988.

[8] Stefik, M. and D. G. Bobrow. "Object-Oriented Programming: Themes and Variations". *AI Magazine*. Vol. 6, No. 4, pages 40–62. Winter 1986.

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

[10] Zicari, R. "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.