Product Models and Reusability*

Frank Feldkamp, Michael Heinrich, Klaus Dieter Meyer-Gramann

DaimlerChrysler AG
Research and Technology
Alt-Moabit 96a
D-10559 Berlin, Germany.

Email: {feldkamp, heinrich, meyer}@dbag.bln.daimlerbenz.com

Abstract

This paper describes and explains the rationale behind the product model that is employed in the configuration design tool SyDeR (System Design for Reusability, [SDR98]). As its name indicates, SyDeR stresses the reuse of system designs in later design projects. The field where SyDeR is applied is configuration design for products for which there is not a complete, fixed set of components, but rather a set of previous designs from which one can be chosen to be reused and/or adapted to the current configuration design problem. This sub-field of configuration design is of major economic importance, as e.g., the design of railway rolling stock or trucks fall into this category.

Importance of reusability

Configuration design takes it for granted that components can be reused in any product configuration. As we had to learn in earlier application studies for configuration design, not every product family meets these requirements. So, it is an important issue to design components and product families in a way that allows to use them in configuration design.

Beyond configuration design in the narrower sense, reusing a component design also saves the time to design it again and the development risks that come with a new development effort. Reuse saves money in development as well as in later stages of the process chain (no new manufacturing or testing equipment is needed, e.g.). Reusability of product and component designs can be significantly increased by intelligent software tools that offer advice on how to reuse and adapt previous component designs as well as to design components for reusability in the first place.

Crucial to reusability is the way the product and its components are modelled. This paper explores how a product model that enables reusability should look like.

Before going into details, we would like to clarify our terminology. There is a large vocabulary for the components of a product (parts, components, assemblies, modules, etc.), which can be a source of confusion. In the SyDeR product model, we decided to call these objects systems. We will stick to this decision throughout this paper and kindly ask the reader to accept this (somewhat abstract) choice for the next six pages.

In the paper, we will first look at a product model that aims straightforward at describing a product without paying much attention to reusability. In the following chapters, we will describe extensions to this simple model that step by step improve reusability.

Modelling product structure as if reusability did not matter

Modelling a product breaks down into three tasks:

- Modelling the properties of a product and its sub-systems,
- modelling the product structure and
- modelling the design logic behind the product.

Modelling the properties is usually achieved by defining attributes for systems describing properties like size, weight, cost or material. The product structure can be modelled by a product tree, a hierarchical bill of materials or a graph (see below). Rules or constraints can describe the dependencies and the reasoning behind the product, thus defining its design logic.

In the following, we will focus on the product structure first and later take a look at how to integrate design logic into the product model. As much as properties are concerned, we will just assume that they are described by attributes of simple (e.g., integer, float, string) or complex data types without going into further details.

The product structure can be analyzed from two perspectives: vertical and horizontal. The vertical product structure is usually modelled by means of a part-of-relationship. Each system is linked to the sub-systems it consists of via such a part-of-relationship. These systems

again are linked to their constituent sub-systems, etc. The transitive closure over the part-of-relationship – beginning with the product root node – yields a product tree.

Besides the vertical structure, the horizontal structure plays an important role in the design of a product, although many product structure models (e.g. in EDM systems) ignore it. It defines how systems are connected and which relationships exist between the systems that belong together building one assembled system. The horizontal structure can be modelled with the help of a connected-with-relationship, leading to graphs which describe the part topology.

Simply connecting systems usually is not enough. Often, there is a need to treat each of the different connections from one system to other systems in a different way. If, for example, a certain system has a mechanical connection with one system and an electrical connection with a third system, you will not want to mix the two connections up. Thus, it is necessary to introduce a modelling construct that allows to make a difference between different connections. A common means for this task is to attach ports to a system. Each port is associated with one connected-with-relationship so that the different connections can be referenced via different ports.

These are also the basic ideas behind the SyDeR product model. It describes systems by listing their properties and the constraints that hold on these properties. It also uses ports to define the potential connectivity of systems. But, as we will see in the following, it adds a few more features that enhance reusability.

A First Step towards Reusability: Types

Part-of- and connected-with-relationships together with ports allow to define the vertical as well as the horizontal product structure in a convenient manner. But at this stage yet, there is not much support for reuse. All the engineer can reuse are instances of systems that have been defined earlier by copying these instances into her current product design.

Following a motto from software enginering, abstraction is the key for reusability. Usually, it is cumbersome to reuse a system exactly as it has been used in a different design context. Too many details that are valid only in that context appear in the system description. This context-specific information can come in the form of specific property values, rules/constraints describing a design logic that only holds in this context, or additional systems which will not be needed in other design contexts.

Instead, the engineer would prefer to reuse a more generalized and more abstract version of the system in question that is not burdened with the specific details of a particular design context. This generic system definition can be provided by a system type, with the actual systems in the different design contexts being instances of this type.

The system type models those properties and rules which hold for all instances of this type. Types allow to reuse system definitions in other design contexts by creating instances of this type in the new design context.

In order to enable reuse of knowledge between types, an inheritance hierarchy of types should be defined. Properties as well as rules/constraints are inherited.

Besides increasing efficiency, type hierarchies also support a least commitment approach to design. The engineer is not forced to commit himself early to detailed design decisions he can not reasonably justify. Instead, in early stages of design, he commits himself to more abstract classes (e.g. to the type „motor“) than in later stages (e.g. to an AC-motor with 360 V).

The inheritance hierarchy also helps to get an overview of the system types available, which is important for the reuse of systems. Usually, the reuse process is semi-automatic. The engineer selects a certain abstract system type (e.g. AC motor) and specifies a few desired properties. A reuse-supporting design tool (like SyDeR) should now search for types which meet these requirements and propose them to the engineer. An inheritance hierarchy helps the engineer a lot to define a starting point for the search (and the tool to search more efficiently).

Bringing Structure and Types Together

So far, we have treated product structure and types quite independently. To closely integrate them, it is necessary to introduce a third kind of entity (besides types and instances) that represents a system used to build another system. Being used in the structural definition of another system *type* (which is in turn supposed to be reused) is a new role that can not be captured by means of types and instances which only deal with reuse of the system itself.

In SyDeR, this leads to *system refinements* (corresponding to types), *system applications* (corresponding to the usage of a system inside another system) and *system instances*.

A similar approach has been used by [Callahan97], based on the ideas of [Rappoport93]. Their part „trinity“ consists of *part*, *part usage*, and *part occurrence*. A part corresponds to a system refinement, a part usage to a system application, and a part occurrence to a system instance. A *part usage graph* defines the (vertical) product structure, while a *rooted occurrence tree* describes the structure of assembly parts.

Modularity by Interfaces as a second step towards reusability

Reusing a system basically means to move it from one context to another. This can only work properly if the system's relationships to its environment (i.e., its dependencies with other systems) are clearly defined and made explicit. A convenient way to model these relationships to the outside world of a system (that means with other systems) is to define interfaces with other systems.

In order to fully exploit the advantages of interfaces, the modelling language for interfaces should be as rich as the one for systems. It should allow to attach properties and constraints to interfaces, interfaces should be typed, and there should also be an inheritance hierarchy for interface types. Types and inheritance yield the same benefit for interfaces as they do for systems. We will explain in the next chapter how useful properties and constraints in interfaces are in terms of modularity.

Interfaces offer a number of advantages:

- Interfaces explicitly describe which relationships exist between a system and its environment (neighbouring modules) and make clear under which terms this system can be reused.
- Interfaces can model connecting parts between systems, relationships between systems (e.g., distance), or even background knowledge about laws of physics etc.
- Interfaces tend to have a longer life-span than systems, and thus provide a stable backbone for the design knowledge base.
- Interfaces are fundamental for modularity, as they decribe the dependencies between systems. Interfaces also help to model the technical reasons for system relationships .

In our application studies for the SyDeR tool, we have used interfaces for a variety of purposes:

- To model the data flow from the initial specification down to the component parameters;
- to model the flow of force along mechanical components;
- to model the relationship between subway tracks and the subway lines ru ning on these tracks;
- to model transportation line for bags in an airport luggage system;
- to model electric connections and control systems in a street car.

Related work has been done by [Heinrich91]. The resource-based paradigm for configuration design can be seen as an ancestor of SyDeR-style extensive interface modelling. In this paradigm, resources also modelled relationships between systems (there called components), but are weaker in terms of expressive power. Properties can be attached to ressources only to a limited extent, and there are only weak constraints.

Constraints should be modular, too

The last step in achieving as much modularity as possible is to modularize not only the product, but also the design logic behind it. This is a crucial factor for the maintenance of the knowledge base behind a configuration design tool, which often suffers from a lack of modularity in the knowledge base. Often, local changes in a knowledge base do not remain local, but lead to changes in other parts of the knowledge base, which in turn cause other changes etc. In order to keep local changes local, the knowledge base has to be modular, too. The importance of this point cannot be overestimated, as the cost for knowledge base maintenance is one of the favourite reasons to terminate the use of a knowledge-based system.

Our thesis is that the direct reference in rules or contraints to properties of other systems causes most of the follow-on changes in a knowledge base. So the key to modular knowledge bases is to eliminate these direct references by using intermediate interfaces.

Constraints attached to systems fall into one of three classes, depending on how spread the parameters/variables in the constraint are:

- constraints over parameters at one single system,
- constraints over parameters at subsystems of one system,
- constraints over parameters which are distributed over several systems.

Constraints of the first kind are local and pose no problem in terms of reusability, as they have no connection to other systems. Constraints over parameters at subsystems occur frequently, e.g. to compute the total weight of a system by summing up the weights of its subsystems. These constraints do not limit reusability if we restrict ourselves to constraints connecting different subsystems of the same system refinement, so that the subsystems of a certain system type are part of its definition and thus available any time the system is used.

The third kind of constraints is more tricky and the one that usually hinders reuse. One cannot assume that several systems are always used and reused together just because

there is a constraint that touches all of them. Moreover, it is not quite clear which system the constraint should be attached to. The solution is to attach it to none of the systems, but to an interface that connects the systems. This has the advantage, that it does not arbitrarily place chunks of knowledge at one system where others are also affected. It also stresses that interface modeling is very helpful in reducing the maintenance cost for a knowledge-based configuration design tool.

Summary

We first introduced a generic product model that was expressive enough to capture product structure, properties and design logic, but did not pay attention to the needs of reusability. We then extendend this basic model into a generic product model that supports reusability as much as possible.

The first step to enhance reusability was to introduce system types and sort them in an inheritance hierarchy. The integration of system types with a recursive product structure lead us to the definition of third entity class besides types and instances that represents the use of a system in the definition of another system *type*. The second step was to introduce interfaces to model relationship between a system and its environment. The third step was to make sure that constraints do not directly reference to objects outside of the system where the constraint was defined.

These steps lead to a product model that has been implemented in the configuration design tool SyDeR. The feasiblity of this approach has been confirmed in several application studies, among them electrical street car design and car engine water pump design.

Related Work

The idea of types is, of course, not a new one. Their integration with a recursive product structure has been solved in a similar way by [Callahan97, Rappoport93]. Object-oriented product models and constraints can, for example, also be found in [deVries97, Ziegler93]. Extensive interface modelling and the strictly modular use of constraints are ideas that we have not seen published anywhere else. We regard the integration of all these ideas and their implementation in the tool SyDeR as the major contribution of this paper.

References

Callahan97
Callahan, S.: „Relating Functional Schematics to Hierarchical Mechanical Assemblies“, Proc. Fourth Symposium on Solid Modeling and Applications 1997, pp.229-239, ACM, New York, 1997.

Heinrich91
Heinrich, M.; Jüngst, E.W.: „A Resource-Based Paradigm for the Configuring of Technical Systems from Modular Systems", Proc. Seventh IEEE Conf. on AI Applications (CAIA'91); 257-264; 1991.

Rappoport93
Rappoport, A.: „A scheme for single instance representation in hierarchical assembly graphs“, IFIP Conference on Geometric Modeling in Computer Graphics, Genova, Italy, June 1993, B. Falcidieno and T.L. Kunii, eds., Springer 1993.

deVries97
de Vries, T.; Weustlink, P.; Cremer, J.: „Improving Dynamic System Model Building Through Constraints"; in: Computer Aided Conceptual Design '97, Proc. 1997 Lancaster Intern. Workshop on Engineering Design CACD '97; Lancaster University Engineering Design Centre 1997.

SDR98
Feldkamp, F., Heinrich, M., Meyer-Gramann, K. D.: "SyDeR – System Design for Reusability", AI-EDAM Special Issue on Configuration Design, September 1998.

Zeigler90
Zeigler, B.P.: „Object-Oriented Simulation with Hierarchical, Modular Models - Intelligent Agents and Endomorphic Systems", Academic Press, Boston; 1990.