Secure Modular Design of Configurable Products

Henk Jan Pels

Secure modular design of configurable products

Dr.ir. H.J. Pels,

Phi Knowledge Process Enabling b.v.
h.j.pels@phi-kpe.nl

Abstract. Design of complex configurable products is a hard task. Testing of all variants is often impossible because the number of variants is orders of magnitude larger than the total life cycle production volume. Modular design is advocated as solution but cannot guarantee that if one variant works OK another will not cause errors. This paper proposes, based on a similar method for modular database system design, a theory for modular design of mechanical systems that poses the concept of module independence, to enable design and test of module families as a separate unit while offering formal conditions to assure that no variant of the module family will cause failures when integrated in the end-product. This allows system verification and testing to be done per module family with no need to test individual end product variants. The method is illustrated with a simple example of a mechanical design, but not yet applied in practice.

Keywords: modular design, product configuration, module independence.

1 Introduction

A configurable product is defined by a number of parameters. By assigning proper values to the parameters a specific product can be configured to fit the requirements of a customer. When one or more parameters control the selection out of a set of interchangeable components, the design is called modular and the interchangeable components are called modules. Product configuration is applied massively in modern industry in order to increase product variety. Especially automotive industry depends heavily on product configuration, such that numbers of product variants within one model range up to 10**20 and larger.

The design of a configurable product is a complex task. Where designing a single product first time right is still a challenge, the design of a consistent product family, with all necessary rules to make sure that every allowed variant will be manufacturable and will work properly, is a huge task, especially because 100% testing is impossible when the

number of variants is orders of magnitude larger than the total life cycle production volume.

Controlling the change process in a configurable product environment is as complex as the original design. Every product family will have numerous changes during its life time. Every change in a module may cause conflicts with other modules. Maintaining the consistency of a configurable product is a huge task. This research proposes a secure method for design and maintenance of configurable products.

2 Modularity Overview

The generic meaning of module is "exchangeable component". The term modularity is used to describe the use of common units to create product variants. It aims at the identification of independent, standardized, or interchangeable units to satisfy a variety of functions (1, 2). For decomposing a system into modules often Design Structure Matrix (3, 4) is used where components with relatively more interactions are combined into one module. Components that are not assigned to any module are called independent. Three types of modularity are distinguished: component swapping, component sharing and bus. Four types of interaction are mentioned by (5): spatial, energy, information and material. Modularity is viewed by (6) as depending on two characteristics of a design: 1) similarity between the physical and functional architecture of the design, and 2) minimization of incidental interactions between physical components. (7) Discusses modularity in the context of integrated process and product design in order to obtain flexibility in terms of increased number of product variants. An approach from the operations management point of view is found in (8). Their qualitative research results in correlations between type of modularity and complexity of production and supply chain. The modular design of production management systems is discussed in (9, 10, 11, 12).

All methods for design of modular products aim at reduction of design cost, but non offers a possibility to prove that two modules are exchangeable in the sense that if one functions without failure in a larger system, the other will as well. Changes in a modular system can be implemented by changing one or more modules and replacing the old versions by the new ones. A method to prove that that if the new module version is consistent in itself, it will also be consistent in the whole system, could reduce considerably the maintenance cost for complex configurable products. This paper discusses such a method for secure modular design.

3 Research question and Approach

When the number of variants is much larger than the total lifecycle production volume, it is unrealistic to test all possible combinations, so it is desirable to have a method to test individual modules independent from the total product, with certainty that all module variants will operate properly in the total system, regardless the choices made for other modules. Also it is desirable to be able to design and verify modules independently, in order to avoid endless design iterations. The problem is that there are many relationships between the components, which may cause failure under non frequently occurring conditions. When there are complex relationships between the components of a product, a change in one component may cause failure in other components. This means that the effort for a change increases with the complexity of the product. If we assume that the frequency of changes also increases with the complexity of the product, we may conclude that maintenance cost will increase with the square of the complexity. This paper describes a method to define module independence as a provable property that enables to design, verify and test a module as a separate entity and yet ensure that the module will not cause failures in other modules during manufacturing or operation. Note that the term module independence as defined in this paper is very different from the term as cited (3, 4) in the previous section.

The research builds upon a modular design methodology developed for large databased information systems (13, 14). The concept of module independence is translated from database systems to mechanical systems and illustrated with an example. This paper explains how the theory can be applied in mechanical design, but does not describe validation in practice.

4 Configurable products and product families.

4.1 Some terminology

The traditional approach in manufacturing is to design a product once and produce identical copies many times. The need for increased variety led to introduction of the concept of product families (15, 16). Understanding product families requires quite some abstract thinking, so precise definition of terms is helpful:

- A **product instance** is a specific, single, manufactured product, as can be touched, sold to and used by a customer,
- A **product type** is defined by a product model, such that each two product instances, manufactured according to this model, are equivalent to the user.

- A **product family** is a class of **product variants**, defined by a number of parameters with specified value ranges. Each choice of parameter values defines a product variant. A product variant may be either a product family itself, or a product type.

Product is used as generic term: it can be an end-product, a sub-assembly or a mono part, and for each of them instance, type or family. A component is a product that is intended to be used as part of an assembly. The implementation of a change in the design of a modular product can be viewed as creating a changed version of a module and exchanging the old module version with the new one.

4.2 Definition of module

As stated above a module is an exchangeable component. That means that in a specific assembly it can be exchanged with another variant of the same product family. When it is not feasible to test the behavior in the end-product of each variant individually, we need formal rules to predict the consistent exchangeability of the whole family. Since a module, in order to be a functional part of a larger system, must have relationships (share space, energy, information or material) with other parts of the system, it is impossible to check consistency of a module without knowledge of some specifications of those other parts. Therefore a module must have interfaces, showing the necessary specifications of related parts in the system. Consequently, in this paper, we define a module as:

> *A module is a product family or product type including the specification of its interfaces.*

This means that a module specification consists of two parts:

1. The **own domain** which holds the specification of the component itself,
2. The **foreign domain** which holds specifications of related components.

The own domain can be split into two parts:

1. The **public domain** with the own specifications that may occur in the public domain of other modules,
2. The **private domain** with the specifications that are hidden for other modules.

Note that in this definition a module interface differs from the usual interpretation of interface as a physical port to exchange or share space, energy, information or material. A module interface is a set of specifications. The foreign domain is the module's model of its environment, the public domain is the model the module shows of itself to its

environment. So in this paper a module interface is defined from the design point of view rather than form an operational point of view.

The essential property of a module is that it can be changed and verified without any other knowledge of the total system than its foreign domain, such that, after verification, it can be integrated in its intended position in the total system, without causing any failures during assembly or operation. This property is called **module independence**.

Supposing that a module type is always a variant of a family of mutually exchangeable components, we call a module independent when:

> *It can be verified and tested to operate properly in the intended product, without knowledge of the intended product other than specified in the module's foreign domain.*

It may be clear that the designer of a module may only change specifications in the own domain of the module. When changing a specification in the public domain, he must check which other modules are using this specification and negotiate with the designers of those modules whether the intended change is acceptable. A desirable, but not self evident property is that the designer can change any specification in the private domain without causing conflicts with other modules. The designer is not allowed to change any specification in the foreign domain of his module, since those are the responsibility of another designer.

4.3 module independence in databased systems

The concept of module independence has originally been developed for data based information systems, which consist of a set of applications that operate on a common database. Maintenance of complex databased information systems appears to be expensive because a change in one application or its database subschema, may cause failures in other applications. These errors cannot always be detected in tests, because it happens that state changes propagate slowly along different elements of the database until a fault occurs in an application that does not even have any data element in common with the changed application. Since any data element may be related, directly or via applications, to any other data element in the total database, every change must be verified against the whole database.

A database is defined by a database schema that specifies object classes, their attributes and constraints that limit the allowed combinations of attribute values. Constraints can refer to single attributes (like "the value must be a positive integer") or to multiple elements (like "this value must be unique within the object class population"). In this way the database schema defines the state space of the database. Applications perform

operations on the database, which can be (1) reading data and performing computational operations on them or (2) writing data in order to add or change data. A fault occurs when an application tries to execute a not allowed state change (like writing a negative integer) or when a value read from the database causes failure of a computational operation (like division by zero). Since the database state space is defined by its schema, it is theoretically possible to check each operation against every possible set of values.

A database module is a subset of object classes and constraints together with a set of applications that operate on only those classes. The subset of element specifications and constraints forms the **module-subschema**. The module state space is obtained by projecting the database state space on the module schema. It is evident that constraints make only sense for a module when all elements referenced in the constraint are in the module subschema. Modules interact through interfaces in the form of shared object classes. Each class is owned by a single module and only applications of the owner module are allowed to write on this object class. This means that the applications of a module may read all elements in its subschema, including foreign elements, but update only own elements. On basis of the usage rights the subschema of a module can be divided in own, private, public and foreign domains, consistent with the definitions above.

Verification of the design of the subschema and applications of a module must make sure that each computational operation is defined for all possible values of the attributes that are referred to and that each change operation results in an element of the state space (meaning it does not conflict with any constraint). It is clear that a change in the design of the foreign domain may cause an own operation to fail on an unanticipated value, while a change in the public domain may cause failure of an operation of another module. The desirable property of module independence must guarantee that each module can be designed, operated and maintained with only knowledge of the module and its interfaces. This property is defined as:

> *A module is independent in a total schema if and only if*
> *every state change that is allowed according to its subschema, is also allowed according to the total schema, and*
> *every element of the module state space is a valid state for all its operations.*

Independence means that a module can be designed, used and changed without risk of failures in the total database, with only knowledge of its subschema. This means that new or changed applications as well as changes in the own domain of the subschema, need to be checked only against the subschema of the module without knowledge of the database structure nor applications outside that subschema.

It has been proven (13) that:

a database module is independent in a database schema, if every constraint in the database schema that refers to an element of the own domain of the module, does not refer to any element outside the subschema of the module.

In practice this means that modules must be designed without using any knowledge that is not specified in the subschema of the module. Note that independence of a module is defined in the context of the total database schema of which it is a subschema. Independence is not a property of a module on its own, but of the module in a specific environment.

4.4 Module independence in mechanical systems

The property of independence as defined above for a database module is exactly the property we wish to have for modules in mechanical systems. If we can extend the concept of module independence from databased information systems to complex mechatronic products, we would have a tool to design modular systems that can be proven correct by proving that each single module is correct. In this paper we extend this concept to the design of mechanical products. However, we expect that the conclusions will stay valid when electronic and other technologies are added.

A mechanical product is an assembly of ultimately mono parts. Some parts can have variable positions because of elastic properties or degrees of freedom in assemblies. A mono part is specified in terms of features like cube or cylinder. Each feature compares to an object class in a database. Each variable of the feature compares to an attribute of the object class and each actual value for a variable to an attribute value. Some variables specify possible movements of the construction during operation (e.g. axle rotation). We call them dynamic variables. When in the part model all static (non-dynamic) variables have singular values, the model specifies a product type. When one or more static variables have a value range assigned, the model specifies a product family. An assembly is defined by a Bill of Materials, where each BoM-line specifies a part with its relative position in the assembly. If the what-used variable of a BoM-line specifies a set of possible parts, it defines a product family based on exchangeable parts.

Constraints in a database schema correspond to constraints in a mechanical design, that limit the range of allowed values of variables. Examples are mutual fit for connected components, design volumes and limits on force, mass, speed, temperature etc. The own domain of the module is the product model itself. The public domain of a module is the list of own elements and constraints that may be referred to (known by) other modules. The foreign domain is the list of elements and constraints of other modules that may be referred to (relied upon) in this module. Private, public and foreign domains together form the **module model**.

Based on this analogy between database systems and mechanical systems, we can define the condition for module independence as:

> *a module of a mechanical system is independent in the system model, if every constraint in the system model that refers to an element of the own domain of the module, does not refer to any element that is not specified in the module model.*

In the following section we will apply this interpretation of a modular mechanical product model on a simple example.

4.5 Example

Let's take the simple example of a metal block (further referenced as Block) with a cylindrical hole and a metal cylinder (Cylinder) that can rotate in the hole (see figure 1). Block has attributes l, w, and h for its outer dimensions. Further is has axis and d for axis and diameter of the hole. The origin of Block is where the axis meets its left side in its point of gravity; the x-axis equal to axis and parallel to the length of Block. Width is parallel to y-axis and height parallel to z-axis. Cylinder has attributes l and d. The origin is the center of the left end. The x-axis is the axis of Cylinder.

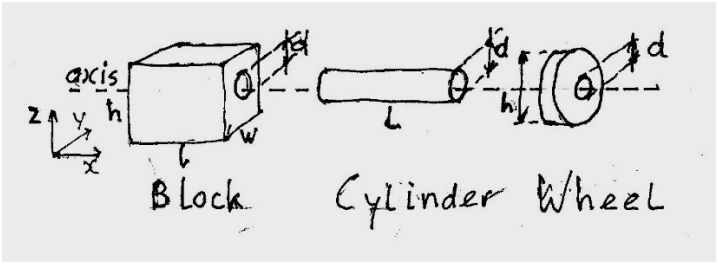

Fig. 1. Mechanical design with three modules

Suppose Cylinder is intended to rotate in the hole of Block. Then the following constraints could apply:

C1. The axes of the hole and the cylinder coincide,
C2. The difference between hole- and cylinder diameter must be between 1% and 2% of hole diameter,
C3. No end of the cylinder is inside the hole.

Because independence is not a property of a module on its own, but of a module in an assembly, the structure of the assembly must be known in order to be able to check independence. Therefore we define assembly Assy with bill of material:

{Part1: Block(0, 0, 0, 0, 0, 0),
Part2: Cylinder(0, 0, 0, rotation, 0, 0)}.

This BOM specifies that the x y and z-axes of Block and Cylinder coincide and that Cylinder can rotate around the x-axis.

In order to be an independent module, the designer of Block needs to know all Cylinder specifications that are referenced in the constraints:

Cylinder.axis, Cylinder.d, Cylinder.l, Part2.

These form the foreign domain of Block and the public domain of Cylinder. Similarly, the foreign domain of Cylinder must specify all hole variables that are mentioned in constraints:

Block.axis, Block.d, Part1.

These become the public domain of Block. Note that Assy, Block and Cylinder are product families, since several variables have value ranges. This makes Assy a configurable product. Constraint C1 is satisfied in the positions of the parts. Constraint 2 an 3 remain as configuration constraints. Rotation is a dynamic variable. Note that in this example the private domain of Cylinder is empty.

Now suppose the system is extended with a wheel, to be mounted on Cylinder, with right planes of Wheel and Cylinder equal. The bill of material line is:

Part3: Wheel(Cylinder.l – Wheel.l, 0, 0, 0, 0, 0).

The following constraints control the correct mounting of Wheel:

C4. Wheel.d = Cylinder.d,
C5. Cylinder.lLength – Block.l – Wheel.l > Wheel.h / 10,
C6. Cylinder.l > 10.Wheel.l.

Because of these constraints the designer specifies the foreign domain of Wheel as:

Cylinder.d, Cylinder.l, Block.l, Part2, Part3.

This makes all relevant constraints ‘visible’ for module Wheel, so Wheel is independent in Assy. However, the Wheel design introduced a constraint that refers to a specification of Block, and thus destroys the independence of Block. This may cause trouble, when configuring a product. Suppose that parameter values for Wheel and Cylinder have been set and a value for Block.l has to be chosen. Since the Block module does not know Wheel, it cannot check constraint C5 and may cause a clash between Block

and Wheel. To restore independence of Block, Wheel.l must be added to the foreign domain of Block.

The example shows how the features and bills of material in the model of a mechanical product correspond to object classes in databases. Feature and BoM variables correspond to attributes of objects. Constraints in a mechanical model limit value ranges of variables like constraints in a database schema limit attribute values. The distinction between static and dynamic variables adds a nuance to mechanical models: static attributes define variants of product families while dynamic variables define the behavior of the product during operation.

5 Discussion

Although the example is extremely simple, it shows how module independence depends on constraints and how ignored constraints cause failures (like when adding Wheel in the example). Constraints are essential for system consistency and thus for managing changes. Therefore it is paramount that all constraints are explicitly specified in the module models and that for every constraint it is clear to which elements it refers. In our experience this is not common in current design practice. Also CAD and PLM systems do not support the maintenance of lists of design constraints, other than, to some extent in the requirements management function. An important task of reviewers is to check for implicit constraints, that are not specified in the design but nevertheless relied upon for any design decisions.

A difference between databased and mechanical systems is that the first have only dynamic variables (attributes), that are used and modified by applications, while a mechanical system has 3 types of variable use:

1. design variables: get their value by design,
2. configuration variables: get their value when configuring a variant,
3. dynamic variables: get their value by operation.

The rules for module independence apply to all of them. When designing a module type, module independence ensures that, although the module has been designed with only foreign domain knowledge of the system, the module will not cause failures when mounted or operated in that system. Simulation should check that dynamic variables do not violate constraints during operation. When designing configurable components, the value ranges of the configuration attributes (also called parameters or options) can be checked against the constraints in only the module schema to ensure that all variants will function consistently in the system. Simulation can be done per module, for a limited range of configuration variable value combinations, which makes a much smaller number of runs than simulating all system variants.

By controlling module independence, a configurable product can be designed module by module. In its design a module must specify its foreign and public domain and all constraints referring to them. An important design decision is how much knowledge the module needs to use about its environment, and what knowledge the module wants to keep private in order to maintain freedom of change and flexibility of the product. Very important is that each module specifies all constraints its design relies upon for proper fit and function. Each constraint must specify explicitly the set of elements to which it refers, as well as the owner module of each element. This specification is the basis for independence checks. Although in the example the public domains are mirrored from the foreign domains of other modules, this is not required. In order to increase flexibility of the design, a module should aim for weak constraints on its foreign domain (allowing a broad range of values) in order to be less sensitive for changes in its environment. At the same time it should aim at constraints on its public domain to be stronger than necessary for its current design, in order to prevent avoidable limitations on future changes. In other words: keep your public state space smaller than your environment currently requires and keep your foreign state space larger. A general rule for modular design is: keep interfaces as simple as possible.

When adding a module to an assembly, it must be checked that:

1. All its foreign elements are specified as public element in the owner module of the element,
2. All its constraints referring to foreign elements, correspond to equal or stronger constraints specified in other modules,
3. All own elements that occur in foreign domain of other modules, are in the public domain,
4. All constraints in other modules that refer to own elements, are visible in the module model, in the sense that all elements they refer to, are specified in its public or foreign domain and are equal or stronger than the corresponding own constraints.

This means that in a modular design every constraint that hits an interface is specified separately in all related modules and need not be identical. Designers may choose to specify constraints for their foreign domain weaker than specified in the other modules and thus verifying for more possible states, in order to be prepared for a broader range of variants in future and to be less sensitive for constraint changes in other modules. Also they may choose to have own constraints on their public elements that are stronger than the constraints they actually show in their public domain, in order to be less sensitive to future changes in the environment.

Changes in a modular design can be:

1. Adding, changing or removing features in a mono component
2. Adding, changing or removing BoM-lines,
3. Changing values for design variables,
4. Changing value ranges of static variables to change the set of possible variants of a product family.

Like in database systems, enlarging the value range of a variable, may cause, after some time of operation, or even after several design changes in other modules, values for variables in other modules, that never occurred before and cause failures that could difficultly be foreseen. Module independence prohibits such failures.

Module independence is a property of a module in a system. So a module that is independent in one system, can be dependent in another system. The bad news is that every change in a system can in principle destroy independence of any module in the system. The good news is that a simple check of the reference domains of constraints is sufficient to ensure that independence is conserved. If a conflict with the independence rule is detected, the designer who's change causes the conflict, has to inform the responsible designer of the other module so that this designer can negotiate what adaptation in which module is best to restore independence.

When the wheel was added as a third module, it appeared that wheel and cylinder were independent, but wheel and block were not, even though there is not a direct connection between them. The designer of the wheel is responsible for this error: he used a constraint that referenced Block, but did not take proper action. This shows how responsibility for system failures can be assigned to module owners, on basis of independence rules: it can be that not the current change is the real cause, but an earlier change in anther module.

6 Conclusion

The research goal was to find a method for secure interchangeability of modules. The method for modular decomposition of large database system (13, 9) has been translated to mechanical design in such a way that the database rule for database module independence could be applied to mechanical module independence. This means that we may consider the proposed method to be proven.

The method enables to design, test and change modules independent from the total system and then check independence in the system using the system list of reference domains of constraints. This check is relatively easy and can even be automated. We believe that this method can substantially reduce design cost for complex configurable

products. However, for practical application of this method CAD-systems need to extended with the possibility to specify interfaces and PLM systems with the function to maintain constraint lists.

A goal for future research is to validate this theory in practice. One of the issues to be researched is how to achieve a really complete system list of constraints since many mechanical designers are not used to document them accurately. Database technology hardly puts a limit on variable values, so constraints only exist if they are specified by design. In mechanical system material properties pose many constraints on value ranges of many variables, not only geometrical, but also in terms of force, speed, temperature, electro magnetic radiation etc. Sometimes such constraints are only discovered in heavy test conditions. This means that mechanical module independence cannot prevent failures because of previously unknown physical phenomena.

References

1. Huang, C. C. and A. Kusiak, Modularity in design of products. IEEE Transactions on Systems, Man, and Cybernetics, A, 28(1), 6677, 1998.

2. J. K. Gershenson , G. J. Prasad & Y. Zhang, Product modularity: Definitions and benefits, Journal of Engineering Design, Volume 30 (2010) issue 3. 295-313.

3. AlGeddawy, ElMaragry, Reactive design methodology for product family platforms, modularity and parts integration, CIRP journal of Manufacturing Science and Technology 6 (2013) 34-43.

4. Jérémy Bonvoisin, Friedrich Halstenberg, Tom Buchert & Rainer Stark, A systematic literature review on modular product design Journal of Engineering Design, Volume 27, 2016 - Issue 7, Pages 488-514.

5. Thomas U. Pimmler and Steven D. Eppinger, INTEGRATION ANALYSIS OF PRODUCT DECOMPOSITIONS, ASME Design Theory and Methodology Conference Minneapolis, MN September 1994, 343-351.

6. Ulrich, K. and K. Tung, Fundamentals of product modularity. In: Issues in Design/Manufacture Integration 1991, pp. 73-79. A. Sharon Ed. ASME, New York, NY, U.S.A.

7. A. Kusiak, Integrated product and process design: a modularity perspective, J. Eng. Design 2002, Vol13, No. 3, 223-231.

8. F. Salvador, c. Forza, F. Rungutasanatham, Modularity, product variety, production volume, and component sourcing: theorizing beyond generic prescriptions, Journal of Operations Management, 20 (2002), 549-575.

9. H.J. Pels, J.C. Wortmann, "Decomposition of information systems for production management", Computers in Industry, Vol. 6, No. 6, 1985, pp. 435-351.

10. H.J. Pels, G.J. Wegter, "Conceptual Integration of Databases for Computer Integrated Manufacturing", in K. Bo, E.A. Warman, L. Estensen: "CAPE'86", proc. 2nd international conference on Computer Aided Production and Engineering, Copenhagen, 20-30 May, 1986, North-Holland, 1987, pp. 455-472.

11. H.J. Pels, "Conceptual Integration of Distributed Production Management Databases", proc. IFIP WG5.7 working conf. on Design, Implementation and Operation of Databases for Production Management, Barcelona may 1989, North-Holland 1989.

12. Pels, H.J., Erens, F.J.,“Dynamic integration: an approach for the design of integrated manufacturing systems.” In: B.E. Hirsch, K.-D. Thoben (ed.), "'One-of-a-kind' Production: New Approaches", IFIP Transaction B: Applications in Technology, North-Holland, 1992, pp 101-110.

13. Pels H.J. Geïntegreerde Informatiebanken, Modulair ontwerp van het conceptuele schema, Dissertation Eindhoven University of Technology, Stenfert-Kroese, Leiden, 1988 (in Dutch).

14. Pels H.J. “Modularity in Product Design”, in Tichem M.e.a. (ed.) Proceedings of the 3rd WDK Workshop on Product Structuring, jun 26027, Delft University of Technolgy, 1998, ISBN 90-370-0169-6.

15. F.J. Erens, The synthesis of variety; Developing Product Families, Dissertation Eindhoven University of Technology, 1996.

16. F.J. Erens , K. Verhulst Architectures for product families, Computers in Industry**,** Volume 33, Issues 2–3, September 1997, Pages 165-178.