CHAPTER 9

An Integration of Feature-Based Design and Consistency Management in CAD Applications

Jivka Ovtcharova, Uwe Jasnoch
Fraunhofer-Institut für Graphische Datenverarbeitung (IGD)
64283 Darmstadt, Germany

Much of recent research on feature-based design has been concentrated on answering such fundamental questions as: What are features? How can they be semantically correct defined? What kind of system architecture is needed to satisfy functional requirements, such as conformability of tools to user requirements and to different applications? Unfortunately, current feature-based design systems do not answer all of those questions, particularly those that deal with the integration of design, reliability and maintainability of constraints. This chapter presents on- going work dealing with design by features using the constraint satisfaction approach. Basic requirements for defining and administering constraints in feature-based models are presented and an architecture for consistency management in feature-based design is proposed. The two main modules of interest is the Feature-Based Design Module and the Consistency Management Mo-dule. The first module is intended to support the design of product parts by creating and manipula-ting feature primitives that possess design semantics. The second module provides functionality for definition, evaluation and satisfaction of constraints in feature-based models.

9.1. INTRODUCTION

Computer-Aided Design (CAD) systems widely used today provide users with facilities to capture engineering semantics of product parts as well as geometry in the part models. Since different types of information have to be represented and manipulated in a natural and expressive manner the *product life cycle* as a whole become more important. This leads to definition of comprehensive models, *product models* based on integrated data.

Recently, *features* have been identified in the engineering community as *meaningful abstractions* with which humans reason about products and processes. From the designer's point of view, features are *functional primitives*, which serve as

R. Soenen et al. (eds.), *Advanced CAD/CAM Systems*

the basis for product representation, improving the quality of the design and the link to life cycle activities, such as process planning and manufacturing.

Recent research on feature-based design has been concentrated on solving such fundamental problems as: How to create comprehensive semantically correct feature-based models? What kind of system architecture is needed to satisfy functional requirements, such as the conformability of tools to user requirements and to different applications? Unfortunately, current feature-based design systems do not address most of these problems, particularly the problem that deal with the integration of design, reliability and maintainability of constraints.

Our approach in addressing most of these requirements focuses on the integration of feature-based design and the management of constraints for CAD applications. We consider the following major issues: 1) establishment of a *uniform definition* for features and constraints using a common formal base, 2) development of *an integrated system architecture* for feature-based design and consistency management and 3) development of *mechanisms* for feature-based design using feature data at different le-vels of abstraction and maintenance of constraints for checking semantically correct feature-based models.

This chapter introduces a system architecture for the integration of feature-based design and the management of constraints. One of the main parts, the *Feature-Based Design Module*, is creating and manipulating features-based product models using *feature primitives* that possess design semantics. These semantics will be automati-cally estimated during the design session using the *Consistency Management Module*.

We deliberately omitted most of the operational aspects during this progress report, and focused on conceptual work which is key for future implementation. At this stage of our investigation, it is important to note that a sound conceptual foundation is both necessary and achievable.

The chapter is organized as follows: Section 2 presents a unified approach to the definition of features and constraints and is based on the main principles of the ontology. Section 3 introduces an integrated architecture for consistency checking in feature-based design. Finally, the conclusion and list of references are presented.

9.2. A UNIFIED APPROACH TO THE DEFINITION OF FEATURES AND CONSTRAINTS

In the following we clarify some basic ideas related to the unified definition of features and constraints using a common formal base. Our emphasis is on the close relation between the definition of features and constraints from an ontological point of view and provide a pragmatic definition of features and constraints in accordance with their modeling capabilities.

9.2.1. Features

Although the literature reflects many different attempts to define features [PRA 88], [SHA 88], [WIL 90], there is *no consensus* on a common precise definition of what a feature is. A wide spectrum of definitions have been presented which only address specific applications. Many authors consider only form (geometric shape) features [WIL 90] but do not include the reason for the functional purpose and usefulness of

the feature. Others focus on the engineering significance of features which can be thought of as engineering primitives. In this sense, Shah gave perhaps the fullest and the most exact definition: "A feature is a physical constituent of a part, is mappable to a generic shape, has engineering significance, and has predictable properties" [SHA 91]. However, this definition does not clarify an *essential difference* between product parts and features because product parts have an existence of their own, while features can only exist within the parts, i.e. they do not have identity without specific parts.

The search for a formal definition of what a feature ought to be has led us to an ontological point of view based on a philosophy of sciences. As described in [OVT 93], *objects possess properties*. Properties do not exist in isolation but are "attached" to objects and can be expressed by *observable attributes* and *laws*. Observable attributes are assigned by *us* to the objects, so that *we* describe already known properties or re-cognize unknown properties through attributes. Laws are constraints on predetermined values of attributes. For example, an essential property of a shaft is to transmit rotational movement. For this goal, several attributes, for example, shape attributes such as a cylindrical basic part, a keyway, etc. are defined by the user to describe this property. Moreover, the shaft is constrained to transmit only rotational, and not translational movement.

Using this fundamental principle we suggest that in the domain of engineering science and practice:

A product emerges as an object and is known through its **properties**. Features are **observable attributes** needed to know the properties of a product and constraints are **laws** needed to define the set of states that a product can really be in.

This notion is formalized in the following definitions:

Definition 1: Let x be a product and $p(x)$ the collection of its properties. Then the product with its properties is called a *concrete product X:* $X=<x, p(x)>$.

The specific set of properties used to describe a product depends on the point of view and the purpose of modeling. This set is called a *functional schema*.

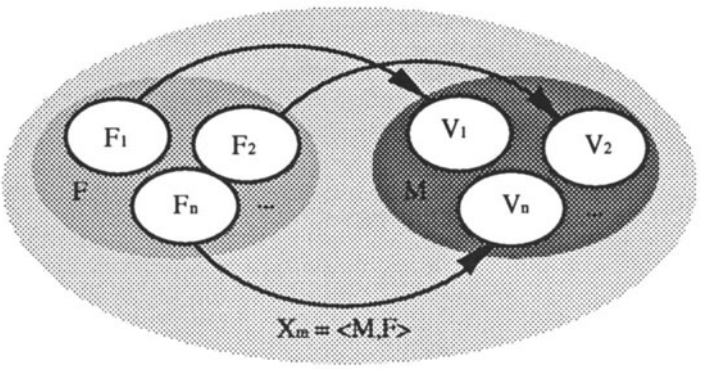

Figure 1. Functional schema of a product X

Definition 2: A functional schema X_m of X is a certain nonempty set M, together with a finite sequence F of functions or values of attributes on M, each of which represents a property of the product. Briefly, $X_m=<M,F>$, where $F=< F_i | \ F_i$ is a function on M and $1 vi vn 1>$.

In this definition M stands for all possible points of view (or models) of a product. In the functional schema (Figure 1), each component is F_i: M≥V_i, where V_i is a domain of values for the property represented by F_i. The F_i are called *state variables* (values of attributes at a certain time), or *state functions*. The set of all values of F_i defines the *state* of the product.

In reality, not every conceivable combination of values of the state function can materialize as a state of a product. Rather, the nature of a product is such that only certain combinations are allowed. The allowed states of a product are determined by the set of product laws. The concept of a law is fundamental in product modeling because it contains the knowledge of what a product can or can not be. A product can only be in states consistent with its laws. Hence, the following definition:

Definition 3: Let X_m=<*M,F*> be a functional schema for a product *X*, where *F:* <F_1 , ... , F_n>:M≥V_1 ... V_n is the state function, and let *L(X)* be the set of all law statements of *X*. The subset of the codomain V_1 V_2 ... V_n of *F* restricted by the conditions (law statements) in *L(X)* is called the *lawful state space* of *X* in the representation X_m, or $S_L(X)$ (Figure 2).

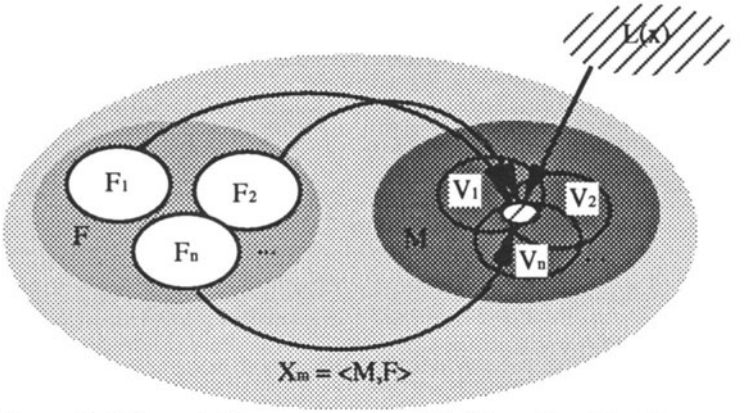

Figure 2. Schematic illustration of the definition of lawful state space

Thus, we propose the following definition of a feature based on the fundamental principles of the ontology:

Features are **attributes** assigned by us to products in order to describe or recognize their properties.

The idea that leads us to the definition of features as attributes assigned by us to products in order to describe or recognize properties is *quite subjective*. A set of specific features is needed to describe a product which dependents on a given point of view for example, on a design and manufacturing point of view and a specific application area, such as a mechanical and building engineering area. For example, consider a product that is designed and manufactured by two different people representing two different disciplines. Each of them may assign different features to it, according to their different views of the product (Figure 3).

Because the application areas and the different points of view to the feature definition are immeasurable, a conclusion has to be that the number of specific

features is not finite. But, following the general definition of features based on the main principles of the ontology, it may be possible to classify them into *main classes* corresponding to the *properties* that they describe. The classification of features is quite useful in the following sense: *first*, grouping features into classes with respect to given properties leads to the development of unified mechanisms for modeling different products possessing these properties. *Second*, feature classification leads to common terminology and development of product data exchange standards.

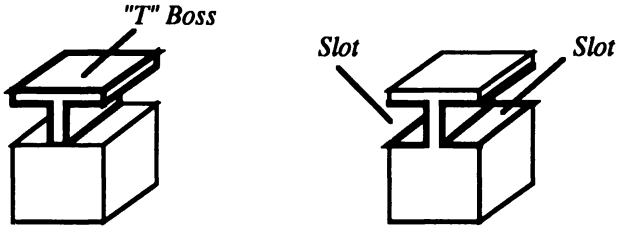

Figure 3. Different views to the same product

Until now, several classification schemes have been proposed. Some of these are based entirely on shape properties [FFI 92], and another on properties specific for gi-ven application areas, such as process planning [GIN 91], tooling cost estimating [ROS 92] and manufacturing [VAN 90]. Each of these schemes is correct in the special scope of its definition.

Our approach to feature classification distinguishes between *generic* and *application* features. Generic features are used to describe properties of products in a general way and not for concrete applications. Application features are defined on top of generic features by assigning data, specific for an application. For example, form is a property of a product that plays an important role in the different phases of the product life cycle. Form can be described by using a limited set of form features (as defined in STEP, Standard for the Exchange of Product Model Data, [FFI 92]). Application features, such as design features and machining features, can be defined by using application-oriented data in a specific context.

9.2.2. Constraints

As mentioned in Section 2.1, constraints describe allowed combinations of the values of features which lead to permitted states of an object (or product). The evaluation of these constraints on state transitions of the product ensures that each state transition ends in a valid product state. The set of constraints can be distinguished between *non-violable* and *violable* constraints. The first set of constraints cannot be violated because it changes the product semantics or leads to an impossible situation. These constraints, such as natural laws or elementary laws describe elementary states, for example, a protrusion cannot have a negative volume. The other set of constraints describes, for example, design policies. These constraints sometimes must be violated, because the design process is iterative.

During the design of a product, the product cannot always be in a consistent state. However, the consistent state for the product must be reached at the end of the design phase. This leads to the requirement of deferred constraints evaluation.

Because constraints describe valid states of a product, the location of constraints has to be known. Figure 4 shows the basic alternatives of possible locations.

The *first alternative* binds the constraint to the *relation between two objects.* This alternative is often used in existing systems. The binding can be strong, or weak depending on the relation between the objects. The basic disadvantage of this alternative is that some constraints, for example, constraints for referential integrity, could not be specified or evaluated, as they do not exist if the relation does not exist. The *second alternative* views the constraint as an *integral component of an object.* The disadvantage of this alternative is the fact that for each object of a class, there exist the same set of constraints. In the *third alternative*, a *constraint exists independently of an object.* The constraint could be applied for all objects of a specific class. This is similar to the real world, where laws exist independently from the objects, for example, persons, whereby the laws are valid for all objects of a specific class.

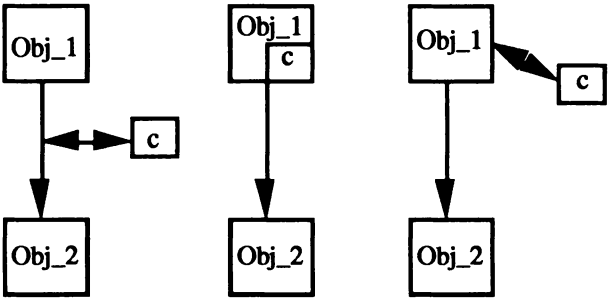

Figure 4. Basic alternatives for the location of constraints

Because, our goal is to integrate constraint checking, in a feature-based design system, we choose the last alternative for several reasons. First of all, we felt that this is the *most natural way* of thinking of and placing constraints. The occurrence of real implemented constraints is reduced, since each constraint is responsible for a set of design features. Additionally, there are mechanisms for administering possible constraints, for example, a parser or an interpreter for an off–line definition of constraints. Although, the mechanism of consistency management could have the capability of handling constraints which are not directly related to design features but nevertheless exist in the system, e.g. for objects in the product data management system. This raises the flexibility of constrains handling. Additionally, this loose coupling between the objects and constraints provides the possibility of monitoring the constraints evaluation, to prevent, for example, endless loops in recursive evaluations. Lastly, this alternative offers a natural way of defining constraints for object communities.

9.3. AN INTEGRATED ARCHITECTURE

9.3.1. Environment

The main goal of the development of a CAD environment for supporting feature-based design is to allow the users to configure the system functionality according to the application requirements. Thus, modifying existing functionalities, integrating new modeling facilities, and incorporating new application tasks should be supported by the system.

Figure 5 represents an architecture of the "whole" application during runtime.

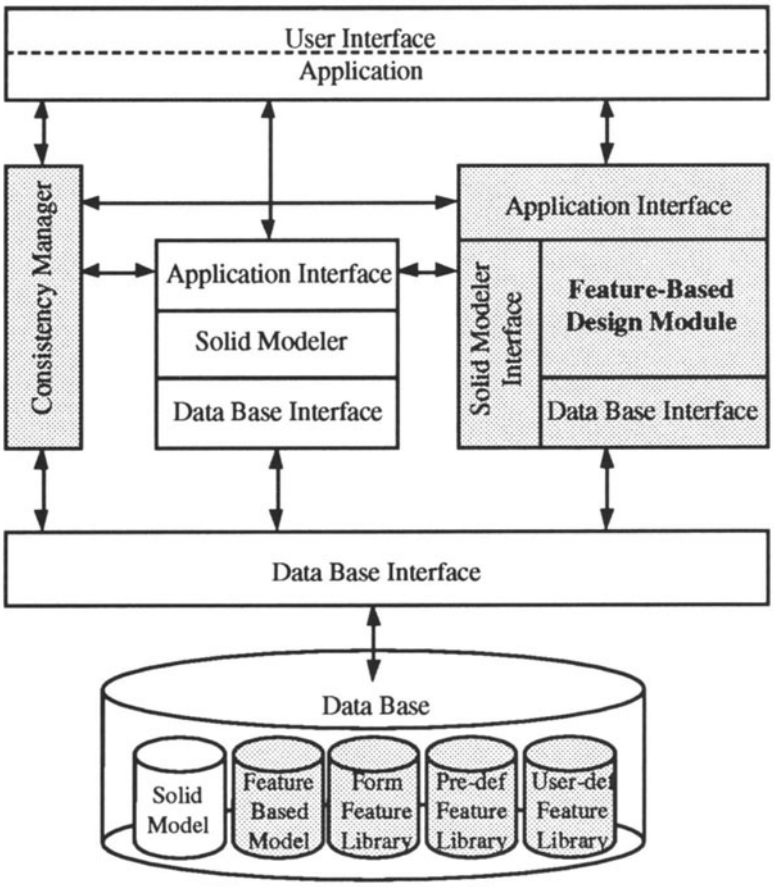

Figure 5. An overall architecture

The main modules can be described as follows: The *User Interface/Application* module is the interface between the application and the user, offers users tools for the interaction with CAD models and allows communication with external systems

and applications, such as NC modules, etc. The *Feature-Based Design Module* allows feature data and design processes to be managed in a uniform way. The *Solid Modeler* is used for evaluation of the feature-based models, initiated by the feature-based design mo-dule. To provide independency between the modelers, the feature-based design mo-dule sends methods to a so called abstract solid modeler. A given solid modeler is then integrated, by implementing an encapsulation shell above its interface, to implement the methods of the abstract modeler. This concepts provides also independency from the underlying evaluation model of the solid modeler, because it is hidden by the encapsulation. The *Consistency Manager* provides services to handle all kinds of diffe-rent constraints within the CAD environment. The *Product Database* includes all services for storing and retrieving various product data. In the following, the focus is on the feature-based design module and the consistency management module within the open CAD environment.

9.3.2. Feature-based design module

The *Feature-Based Design Module (FBDM)* is responsible for creating and manipulating features-based product models using *feature data at different levels of abstraction*. The FBDM follows the top-down and bottom-up approach of design using a hierarchical modeling scheme, which make it possible to trace from design features representation (application-oriented) to generic features representation (form, material, etc.), and finally, to a solid representation [OVT 92]. The idea is that the design features defined on the top level and instanced from the feature library are used to model a product part in a concrete application context. Their description derives specific meaning from the view of the function of the product part, including shape data as well as non-shape data. Moreover, the feature-based design module supports the users to *define their own specific design features*, thereby allowing the representation of new types of products. This is accomplished by a design that foresees the interactive specification of design features by the user. The starting point of the design process are *pre-defined design features* delivered with the system. During the design process it is possible to combine and store them as a new *user-defined design feature*. By repeating this process more and more complex design features are immediately avai-lable in the process of design. Furthermore, the design foresees the *interactive definition of constraints* concerning the semantics of the model. Constraints include rules to express the relationship between feature parameters, like radius and height, or feature elements, like feature faces and edges. The specification and administration of constraints, as well as their dynamic evaluation, is defined and controlled by the *Consistency Management Module* (see Section 3.3).

In order to realize a feature-based design module as independent as possible from the concrete CAD environment, the FBDM is equipped with external interfaces to the user interface, to the underlying solid modeler and to the database.

9.3.2.1. Feature libraries

The feature libraries are used to set-up the context for the design in the feature-based model. Here, two types of design feature libraries are distinguished: (1) pre-defined design feature libraries which are always available and define a fixed set of most used design features, and (2) user-defined feature libraries which complete the

previous feature set and are dynamically extended during the design session.

The set of pre-defined design features consists of standard features, such as cylindrical holes, rectangular pockets, simple slots, etc. Each pre-defined design feature is implicitly described by a name, the type of the corresponding form feature (depre-ssion, protrusion), a list of design parameters, constraints on these parameters and a set of methods which are necessary for creating and manipulating the feature. Thus, product parts are modeled by instantiating features from the library.

In Figure 6 an example of pre-defined design features 'screw hole' and its corresponding form feature is shown. An instance of such a hole is created by calling the method create_instance_object () with an assigned value for each parameter. Transformations and modifications of the feature model can be provided by the method manipulate_instance_object (), while the consistency checking is performed by the method validate_instance_object ().

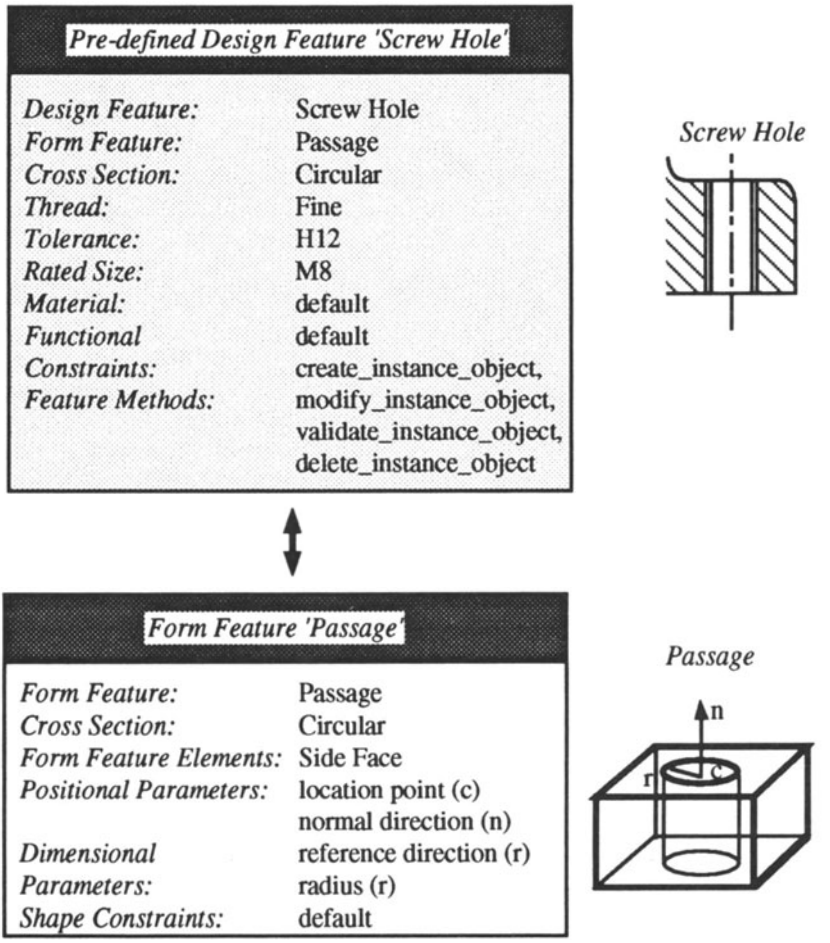

Figure 6. Pre-defined design feature description: an example of a screw hole

If the set of pre-defined design features is not sufficient, designers can specify their own user-defined set of features, which will be included in a user-defined design feature library. User-defined design features can be created by means of the feature mo-deler or of the solid modeler, depending of their shape complexity. The feature mode-ler is used to describe user-defined features obtained from a composition of pre-defined design features. The stepped boss described in Figure 7 is an example of a feature defined by the user through standard design features.

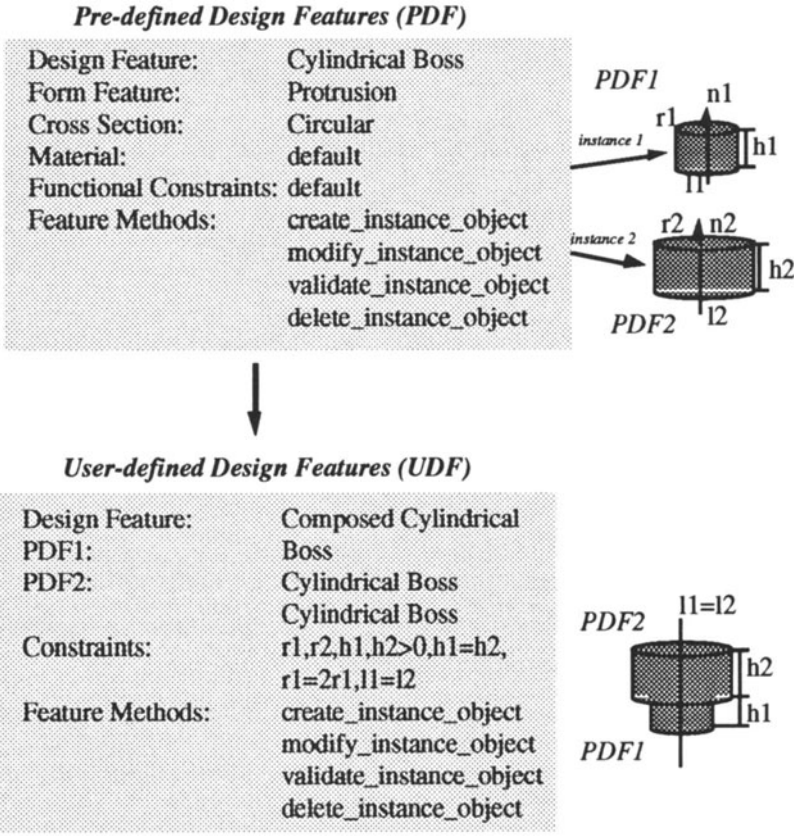

Figure 7. User-defined design feature description: an example of a composed cylindrical boss

In the case of features having a complex shape, the users can create their own features by means of the solid modeler and must define the feature volume which affects the main shape of the part. The shape feature is then explicitly represented by the geometric model but the information describing the type of feature is not complete as for pre-defined design features: implicit representations are not considered and manipulations cannot be provided. Users can only associate a type, an identifier, a position and a set of technological attributes to the part.

9.3.2.2. Feature design process

The user can design a part by instantiating design features from the library. Then, a feature-based design graph is created, where nodes correspond to the design feature instances and arcs store the spatial relationships between them. In the next step, design features are geometrically evaluated using a boundary processor. The boundary evaluation process consists in traversing the feature graph structure. A graph traversal is used for searching for each node in the graph and initiating the corresponding boundary evaluation of the feature. The created volume is defined using a solid modeler offering geometric modeling methods such as sweeping, ruled surfaces, etc. More detailed description of the boundary evaluation process is given in [DEM 94].

9.3.3. Consistency management module

The Consistency Management (CM) concept of constraint handling is based on the CRUX concept introduced in [KÖH 92]. The usage and extensions of the CRUX concept are shown in [CAD 91], [DAS 92], and [JAS 92]. The mechanism provides functiona-lity for the *definition*, the *evaluation* and the *satisfaction* of constraints, according to the general behavior of consistency mechanisms [KÖH 92], [JAS 92]. For automatically checkable constraints (e.g. by a product database management system) constraints can be *enabled* or *disabled* to switch such checks on and off, due to several policies or design states. The same functionality also exists for the rules inside a constraint. Disabled rules stay disabled after a disable-enable sequence of the related constraint. If an explicit evaluation of constraints is required, several *check* operations provide this functionality for the evaluation of the related rules via the procedural interface.

The constraints are specified through a set of rules for objects, classes, tools, or any other entity of the data model or the framework and, should be viewed as objects by themselves for the management of constraints [OVT 93]. Therefore, objects can be created independently from the application. This will be realized by an interactive interface for the definition of the constraints. For dynamic definition of constraints du-ring runtime of the application, a set of procedures is used to integrate the new constraints into the consistency manager.

The basic CRUX mechanism of [KÖH 92] was extended in some parts (see also [JAS 92]). An important extension is the capability of building hierarchies of situations. This supports the aspects of grouping and ordering for administering and termination of constraint evaluation via cycles detection. A situation itself is the main entry point of the data structure of the consistency manager. As stated in [OVT 93], constraints are defined on state transitions of an object. These state transitions define events, here so called situations, in the system, the entry_transition and the exit_transition event. These situations are monitored by the consistency mechanism and provide the basis for actions. According to the two notifications, the situations can be forced to start the evaluation of constraints on the entry_transition, the exit_transition, or on both transition notifications. For each situation there exists an arbitrary number of constraints, at least one constraint. The possibility of binding more than one constraint to a situation will decrease the number of necessary situations in the system and makes situation monitoring easier. In addition, there

exists a mechanism which allows the cluste-ring of constraints to a certain user, a group of users, a project, tasks etc. which corresponds to the different user groups of a system. Clustering of constraints will increase the evaluation performance as the number of related constraints for the evaluation is reduced.

Figure 8 presents the idea of the hierarchy of situations with related constraints. Si-tuations are symbolized by boxes and capitalized letters while constraints are symbo-lized by circles and lower letters. Thick lines indicate the hierarchy between situations while thin lines represent the related constraints. The underlying light dotted area indicates possible user clusters while the middle dotted area symbolizes possible tasks or group clusters.

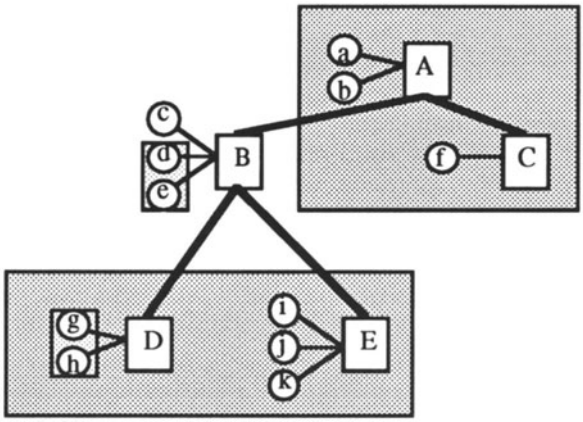

Figure 8. Hierarchy and clustering of situations and constraints

Another important extension is the abstraction of a rule. As state before, the rule express the consistency. Due to the different aspects of consistency, different rule (constraint) solving techniques exists, e.g. numerical techniques or predicate-based solver. Therefore, it is necessary to provide the possibility for accessing different sol-ving algorithms to tackle the different aspects of problems and to provide a most natural way for defining a constraint. This concept solve this task by defining an abstract class *rule* with several derived subclasses for the different kinds of solving techniques. Thus, every instance of the class *rule* has the knowledge of the corresponding solver. Every subclass inherits three methods of the superclass to ensure an uniform access for each instance by the Consistency Mechanism, namely *define, delete*, and *evaluate*. This extension has the benefit of providing the integration of different solving mechanisms into one homogeneous environment. Also, the concept is extensible for new solvers, because for each solver a new subclass will be introduced to make solver the accessible.

The functional interface allows tools to define dynamically new constraints. The scope of these constraints is normally the scope of the local tool, here the *Feature-Based Design Module*. A tool has the possibility to explicitly raise a situation, either in the local tool scope or in the global scope. This capability allows the implementation of "background" tools, which are able to raise specific situations like "whenever machine load is low" or "at midnight" which cause complex and expensive constraint evaluations. This is usually the case at the end of a design

phase, where for example several company policies have to be ensured.

The architecture of the Consistency Module, the Consistency Manager, is divided into several components as shown in Figure 9. The solid relations between several parts indicate, that there could be communication between these related components. The dotted lines inside the *Consistency Manager* symbolize signals. The light shaded box represents the interface between the internal components of the *Consistency Ma-nager* and "clients" of it, while the dark shaded box acts as an internal interface to a "fictive" data base implementation where the constraints and rules and their states (enabled or disabled) are stored persistently. The behavior of the Consistency Mana-ger is shown by example in the section of the runtime-environment.

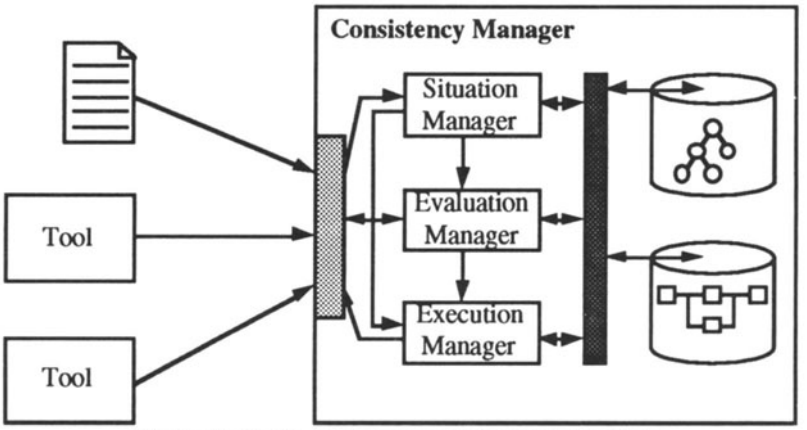

Figure 9. Basic consistency manager architecture

9.3.4. The runtime-environment

This section focuses on the runtime-environment of the application, including the feature-based design module, and the consistency mechanism. Figure 10 presents an example of the runtime-view.

Initially, the situation manager of the consistency mechanism retrieves the relevant situations out of the database, which stores the necessary data of the consistency mechanism. This is done once, in the start-up phase of the application.

As state in Figure 10, each object sends notifications for a state transition of the object, the entry notification and the leave notification (1). In this example, the notifications are send by the *moveSlot* method. These notifications will be monitored by the situation manager by the *MonitorSituation* method. This methods determines the relevant situations. If there is no situation defined for the current event, the situation ma-nager does not interrupt the execution of the application method (8). If the methods detects relevant situations, it send the set of related constraints to the evaluation ma-nager (2). The evaluation manager apply on each enabled constraint the method *CheckConstraint*. This method apply forces an *evaluateRule* on each enabled rule of the constraint (not shown in the figure). This method computes the evaluation, depending on the underlying solving mechanism.

If all rules of all constraints are satisfied, the evaluation manager (7) and the situation manager (8) returns. For each violated rule the executions manager is called (3). He executes the method which was attached by the user in case of a violation by the method *ExecuteMethod*. Executing the method may call another application method (4)(5). This may lead to a recursive actions in the system. It is the task of the situation manager to detect possible endless loops. After executing the method, the execution manager returns back to the evaluation manager (6). After the executing all methods of violated rules, the system is in a consistent state according to the defined constraints.

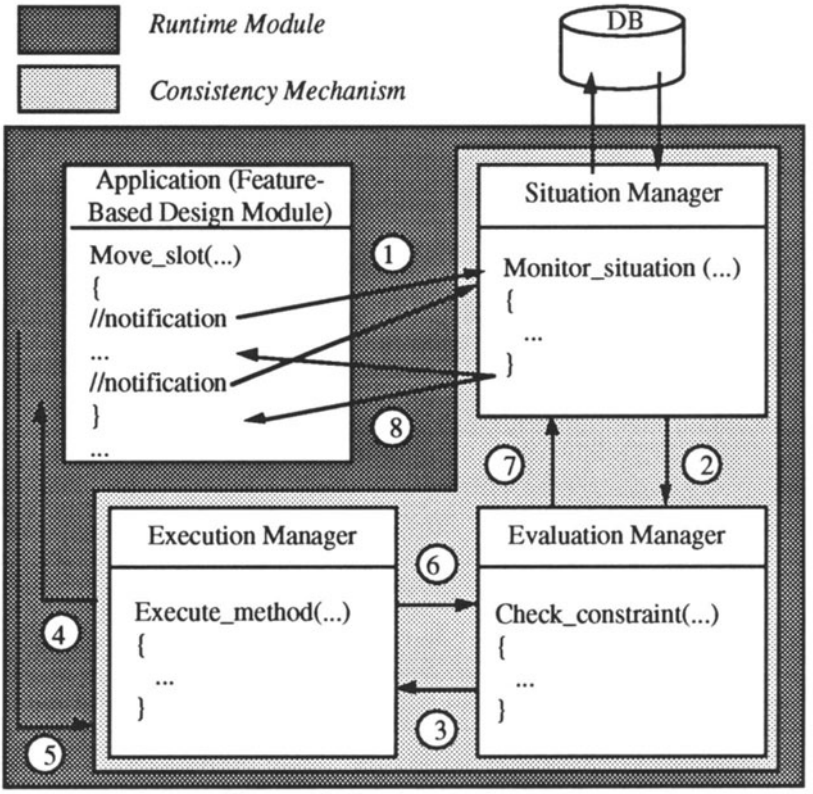

Figure 10. Runtime-environment

9.4. CONCLUSION

The content of this contribution present the definition of features and constraints in a uniform way using the main ontological principles. Based on the observation that features are attributes, and constraints are laws for defining and manipulating properties of objects, we can conclude that for supporting CAD applications feature-based design must be integrated with consistency management. In this sense, we have proposed an integrated architecture for consistency management in feature-

based design systems. Two main modules, the Feature-Based Design Module and the Consistency Management Module have been presented in details. At this stage of our investigation, we do not provide an overall solution, but demonstrated that a sound conceptual foundation is both necessary, achievable and realizable. The first realization of a prototype Feature-Based Design Module and a prototype Consistency Management Module is already implemented in the C programming language. Currently, a new object-oriented version of both modules (C++ programming language) are in development and will run under the client server model X Window on SUN SPARC station. The dialog management is supported by OSF/Motif.

9.5. REFERENCES

[AND 90] ANDERSON D.C. et al., "Geometric Reasoning in Feature-based Design and Process Planning", Computer & Graphics, Vol.14, No.2, 1990, pp.225-235.

[BUN 77] BUNGE M., "Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World", *Reidel*, Boston, 1977.

[BUN 79] BUNGE M., "Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems", *Reidel*, Boston, 1979.

[CAD 91] CAD FRAMEWORK INITIATIVE, "Consistency Management". Version 0.5, Boulder, CO, USA, 1991.

[DAS 92] DASSY PROJECT, "DaDaMo: The DASSY Data Model", Version 2.0, Darmstadt, Germany, 1992.

[FFI 92] "Industrial Automation Systems - Product Data Representation and Exchange - Part 48: Integrated Generic Resources: Form Features", Working Draft, *ISO TC 184/SC4 WG3 N102 (P5)*, January 2, 1992.

[GIN 91] GINDY N.N.Z. et al., "Product and Machine Tools Data Models for Computer Aided Process Planning Systems", In: G. Doumeingts, J. Browne and M. Tomljanovich (Eds.), *Computer Applications in Production and Engineering: Integration Aspects*, Elsevier Science Publishers B.V. (North-Holland), IFIP, 1991, pp.527-534.

[JAS 92] JASNOCH U., "Global Consistency Management within a CAD Framework", In: Proceedings of the *Third IFIP WG 10.2 Workshop on Electronic Design Automation Frameworks*, Paderborn, Germany, 1992.

[KÖH 92] KÖHLER D., "Konsistenzsicherung in graphischen Anwendungen - Modellierung und Sicherstellung graphischer Constraints mit dem CRUX-System", PhD thesis, Technical Univesity, Darmstadt, Germany, 1992.

[OVT 92] OVTCHAROVA J., "A Hierarchical Data Scheme for Feature-Based Design Support", In: *VDI Berichte* Nr. 993.3, 1992, pp. 33-47.

[OVT 93] OVTCHAROVA J. et al., "Towards a Consistency Management in a Feature-Based Design", In the *Proceedings of the 13th International Computers in Engineering Conference CIE'93*, San Diego, CA, August 8-11, 1993, pp. 129-143.

[DEM 94] DE MARTINO T. et al., "Feature-Based Modeling by Integrating Design and Recognition Approaches", to be published in CAD, 1994.

[PRA 88] PRATT M.J., "Synthesis of an Optimal Approach to Form Feature Modelling", In: Proceedings of *1988 ASME International Computers in Engineering Conference and Exibition*, Vol.1, ASME, New York, 1988, pp.263-274.

[ROS 92] ROSEN D.W. et al., "Features and Algorithms for Tooling Cost Evaluation in Injection Molding and Die Casting", In: Proceedings of *1992 ASME International Computers in Engineering Conference and Exibition*, Vol.3, ASME, 1992.

[SHA 92] SHAH J. et al., "Mapping Design Features to Machining Features", In: Proceedings of the *Fourth IFIP WG 5.2 Workshop on Geometric Modeling in Computer-Aided Design*, Rensselaerville, New York, USA, September 27 - October 1, 1992.

[SHA 91] SHAH J.J., "Assessment of Features Technology", *Computer-Aided Design*, Vol.23, No.5, June 1991, pp.331-343.

[SHA 88] SHAH J.J. et al., "Functional Requirements and Conceptual Design of the Feature-Based Modelling System", *Computer-Aided Engineering Journal*, February 1988, pp.9-15.

[VAN 90] VANDENBRANDE J.H., "Automatic Recognition of Machinable Features in Solid Models", PhD thesis, University of Rochester, Rochester, New York, 1990.

[WAN 89] WAND Y., "A Proposal for a Formal Model of Objects", In: W. Kim, F.H. Lochovsky (Eds.), *Object-Oriented Concepts, Databases and Applications*, ACM Press, 1989, pp.537-559.

[WIL 90] WILSON PR., "Feature Modeling Overview", In: Proceedings of the *17th International Conference On Computer Graphics and Interactive Techniques SIGGRAPH'90*, Course Notes 12, Solid Modeling: Architectures, Mathematics, and Algorithms, Dallas, August 6-10, 1990, pp.XI-1 to XI-56.