Published as: Fridqvist, S. and J.P. van Leeuwen. 2002. “Feature Type Recognition – implementation of a recognizing feature manager.” In: *Distributing Knowledge in Building*, Proceedings of CIB W78 2002, Aarhus, Denmark: 12 – 14 June 2002.

Title: **Feature Type Recognition – implementation of a recognizing feature manager**

Authors: Sverker Fridqvist and Jos van Leeuwen

Institution: Eindhoven University of Technology
Faculty of Architecture
Design Systems Group

E-mails: S.Fridqvist@tue.nl
J.P.v.Leeuwen@tue.nl

Abstract: *This paper reports the first phase of a research project to implement and apply Feature Type Recognition (FTR). This technology has many potential areas of application, such as case retrieval, product finding, translating models between schemas, and certain types of analysis.*

Feature Type Recognition is part of the Internet based design knowledge sharing system developed at the authors' department. The system allows communication of highly abstract concepts as well as concrete data. Additionally, it supports a layered approach to modelling, which will facilitate standardisation efforts.

Feature Type Recognition is the process of finding feature types that correspond to a specific feature instance. The paper shows how feature based modelling creates a foundation for feature type recognition. Additionally, it presents and discusses how the Recognizing Feature Manager has been implemented.

Application of Feature Type Recognition to building product finding will take place in the second phase of the research project. However, the paper already discusses basic principles of how this can be done.

Keywords: *Modelling; feature type recognition; design support; property-oriented modelling.*

Introduction

Feature Type Recognition (FTR) is the ability of Feature Based Modelling (FBM) to recognize what *feature types* might fit a particular *feature instance*. Thus, a part of a design modelled as feature instances can be analysed in terms of previously defined feature types. This can be used for many purposes, one of which is to find building products that would fit in a particular design.

The *feature manager* is the central part of the application that takes care of maintaining the database and keeping it self-consistent according to the rules of FBM.

Feature based modelling is a modelling framework developed by Jos van Leeuwen at Eindhoven University of Technology to support design at early stages [van Leeuwen 1999]. It achieves modelling of generic design concepts through feature types, and of individual design objects through feature instances.

FBM is *property oriented*, i.e. the basic modelling objects of FBM represent properties of things in addition to the traditional classes of things [Fridqvist 2000; van Leeuwen, Hendrix, Fridqvist 2001]. Thus, the designer can let the model develop as required by the design, by adding properties to the design model as they become known.

In FBM, feature types are defined in terms of other feature types. Thus, high-level concepts are defined in terms of simpler ones. The user can define new feature types to model whatever concept he finds useful. Nevertheless, also user-defined feature types can be successfully communicated, provided they are based on feature types that are already known to both communicating parties.

Since FBM defines a generic modelling schema, it is able to represent a wide range of concepts, including buildings and building parts. Furthermore, it supports modelling functionally defined things as well as things defined by their composition or other attributes. This enables the user/designer to define the design in terms of function at early stages and to add information about the specific implementation later on.

The possibility to specify things functionally combined with FTR opens an interesting range of possibilities. Given libraries of suitably defined feature types, it should be possible to:

- Find previously stored designs that satisfy a set of requirements (a.k.a. case retrieval).
- Find products that satisfy a set of requirements (a.k.a. product finding).
- Translate models based on one schema to another schema.
- Determine if a design proposal satisfies specified requirements.

The research project reported in this paper has implemented FTR and applies it to product finding, as a demonstration of its applicability.

DesKs

The research project reported here has a parallel project that aims to develop *Design Knowledge Servers*, DesKs [van Leeuwen and Fridqvist 2002]. This project aims to develop a system of networked servers and clients that manage and distribute design knowledge between designers. The rationale of the system is to support collaborative design, but it will also serve other purposes such as creating repositories for public design information. It could thus serve as the "building product information gateway" to serve designers with product information, which has been described by Augenbroe [1998].

Modelling products to support product finding

To serve as an example for evaluation and demonstration, a small product database will be implemented in the FTR-enabled DesKs application. This belongs to the second phase of the research project, thus conclusive results are not expected until later this year (2002). Nevertheless, some requirements that product finding puts on FTR are presented here to highlight some of the complexities of implementing FTR in the context of feature based modelling. In particular, to successfully implement FTR does not only involve writing the FTR program code, but also to study how to use the FBM framework for modelling.

To support product finding, the *functional properties* of a product need to be modelled as well as other properties such as shape, material, colour etc. The FBM framework supports the "rich product semantics that provide complete product models with embedded links to relevant codes and regulations, specifications, geometry, assembly instructions, etc." that will be the foundation of electronic product catalogues [Jain and Augenbroe 2000]. Such catalogues will, according to these authors, "offer added capabilities such as: dynamic updates; sophisticated search capabilities based on performance criteria, availability; multiple information sources; customisation based on user/firm preference. "

Feature Based Modelling

The following part is a brief explanation of some important parts of the FBM framework; for an in-depth description, the reader is directed to other accounts of the FBM framework, in particular [van Leeuwen 1999]. The text here focuses on constraint types, which are treated only briefly in earlier texts.

FBM models generic concepts through *feature types*, and particular individuals through *feature instances*. Thus, a model consists of a collection of interrelated feature instances, which refer to generic concepts modelled as feature types.

Complex features are defined trough components

In the FBM framework, features bearing higher-level meaning are created by combining lower-level ones in a structured manner. *Complex features* combine other features, both at the type level and at the instance level, through *components* that connect higher-level features to lower-level ones. Components are *named*, and can have one of three *role types* that define the relation between the higher-level feature and the lower one, i.e. decomposition, specification or association. A component thus defines what *role* the lower features play in the context of the complex feature.

Additionally, at the type level components include *cardinality* information, which defines the lower and upper limits of the number of instances that, at the instance level, should be related through one single component.

Subtype-supertype hierarchies

In addition to components, feature types may be arranged in subtype-supertype hierarchies. The subtype *inherits* all components of its supertype, but it may re-define them to be more specific. Similarly, a subtype inherits the constraint assignments of its supertype (see next part).

Constraints

Constraint feature types define constraints on a generic level. They include a list of typed parameters, and an *expression* that defines the actual constraint.

Constraints serve two different but related functions; at the type level, when used in a complex feature type definition, they form part of the semantic content of the complex feature type. At the instance level, constraints are used to evaluate the model; the result is either *fulfilled* or *not fulfilled.*

Constraints used to define semantic correlation between components

Presently, two uses have been established for constraints in complex feature types. The first is to define a semantic correlation between components. FBM supports defining tree-structured hierarchies of feature types, with a generic type at the root and more specific types at the nodes. Often, it might be desirable to define a component with a cardinality > 1 for a generic type, while more specific types need to name one of the *fillers*[1] separately through a specific component, see the example in Figure 1. However, the filler should fill both roles, both the generic one and the specific one. To express this kind of inter-component relationship, a new construct has been added to the original FBM framework as defined in [van Leeuwen 1999]. The addition is similar to the component, and a complex type may have several such *constraint attachments*. A constraint attachment specifies a constraint type, and links the *parameters* of the constraint type to the components of the complex type.

The usefulness of constraints is shown by the example illustrated in Figure 1. The rounded boxes are feature types; triangle-ended lines are supertype relations; circle-ended lines are decomposition relations:

- The feature type *Vehicle* is defined as having a component *wheels*, which defines that the vehicle has at least one *Wheel* as parts.
- A *Bicycle* is a subtype of Vehicle and *inherits* the wheels component from its supertype Vehicle. Bicycle additionally has two components of its own, *frontWheel* and *rearWheel*, both of which are of type *BicycleWheel*.
- BicycleWheel is a subtype of Wheel.

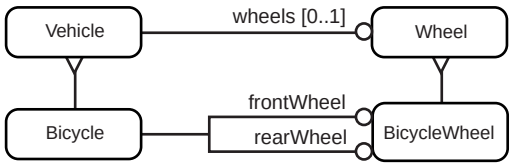

Figure 1 Illustration to show a use for constrain (FBM graphical notation)

Since the current version of the FBM graphical notation has no means to represent constraint expressions, the constraints that define how the components are semantically related for the type are not visible in Figure 1. However, there would need to be three constraints to correctly define the Bicycle feature type:

- Any instance at frontWheel *is also* an instance at wheels, i.e. a front wheel is one of the wheels.
- Any instance at rearWheel *is also* an instance at wheels, i.e. the rear wheel is one of the wheels.
- Any instance at frontWheel *is not* an instance at rearWheel, i.e. there should be two distinct wheels.

Constraints used to model the structure of things at the type level

A second use for constraints in complex feature type definitions is to model a thing’s *structure* at the type level. The ability to define the structure is necessary for modelling functional properties, since these are part of the structure. The structure of a thing can be parted in the internal structure and the external

[1] A *filler* is an instance that is linked to another instance through a component. In other words, the filler *fills the role* specified by the component.

structure, where the internal structure is the complex of relations among the parts of the thing, and the external structure is the complex of relations between the thing and its environment [Ekholm & Fridqvist 1996].

To highlight how constraints are used to define the structure of things, we will use the following example, illustrated in Figure 2. In this case, we want to define a type that would be instantiated as to the right of the figure; the corresponding type is illustrated to the left.

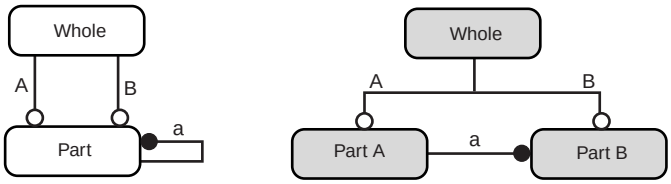

Figure 2 Modelling structure in FBM; type level to the left and instance level to the right

The reader should note that the feature type **Part** to the left only defines that the component **a** should be filled by *some* instance of type **Part**, *not what particular instance*. Apparently, adding the two components **A** and **B** to the type **Whole** is not sufficient to ensure the desired outcome. The necessary additional piece of information can be contributed by adding a constraint at feature type **Whole**, to constrain the sub-component **a** of component **A** to be *identical with* component **B**.

As we can see, using constraints to define structure requires constraint expressions to be able to access the interiors of components. Support for this has been included in the implementation.

Implicit components

It should be noted that feature types serve two different but related purposes. Firstly, they are templates for creating instances. Secondly, a type defines the semantic meaning for any instance that complies with its definition, whether it was initially instantiated from the type or not.

Because of the first fact, it is often not necessary to instantiate all components of a feature. Only if the user wants to differ from the definition of the type, from the *default*, components need to be created. This is utilised to capture the user's conscious moves. All features and components in a model have been actively put there by the user, and thus represent conscious user design moves.

However, in some cases the components prescribed by the type are needed. A common occasion for this is when the user tries to inspect the details of his model, below the level where he has explicitly defined components. Another example is if the model is to be evaluated; then the evaluation application may need these data to perform its activity. To supply the data, the manager has a mechanism to provide *implicit components*, i.e. components (and their fillers) prescribed by an instance's type but not explicitly created by the user.

An additional benefit of the implicit component mechanism is that it allows the user to model at a high semantic level, since it can provide all details defined by the types when necessary. It also makes automated schema translation possible, by providing the basic components that are the common basis for translation.

Creating implicit features may involve inter-relating them with other features, to provide the structure of the thing modelled by the feature. For this purpose, constraint feature types are used to determine what actual feature instances should be related, and through what relations.

Feature Type Recognition

While a feature instance is created based on a specific feature type, the user is free to add to and change the components. Thus, after a while a feature instance may no longer be a true instance of the original type. As a result, the higher-level meaning of the instance may have become hidden, i.e. it is not explicit, but only implicit in terms of the combination of its lower-level concepts as defined by the components. To make the meaning explicit, a higher-level concept needs to be found that represent the actual meaning. This is achieved through feature type recognition (FTR).

The FTR function can be used for other purposes. It can be initiated either by the (human) user, or by some application. For the user, FTR can clarify the meaning or interpretation of a model that has undergone many changes, as said above. It can also suggest further development of the design by

showing various ways to make a general model more specific, or vice-versa. This functionality is utilised in the application part of the research project that runs in 2002.

For applications, FTR can support analysis of models by re-classifying model features according to type libraries defined for the purpose of analysis. This functionality might also be used to translate models between different modelling schemas, and thus support data exchange.

How FTR is implemented

FTR finds the types that best cover the present status of a feature instance by comparing the components of the instance with the components of the types. The FTR algorithm has two phases. The first is to find applicable *candidates*, i.e. feature types that might be an appropriate type for the instance. The second phase is to select one or several of the candidates. To accomplish this, the candidates first are sorted according to applicability.

Finding candidates

In short, if a feature instance *A* has a component with an instance *a* of a specific type *t*, the instance *A* may be recognised as the type *T*, which has *t* as one of its components. The process of finding candidates proceeds through the following steps, as shown in Figure 3:

1. Inspect all the *component features*.
2. Collect the types of all *component fillers* (2.1) as well as the fillers’ supertypes (2.2).
3. For each type t from step 2, collect the types where t is a component. The result is the list of *candidate types*.
4. For each of the types from 3, add the subtypes to the candidate list.

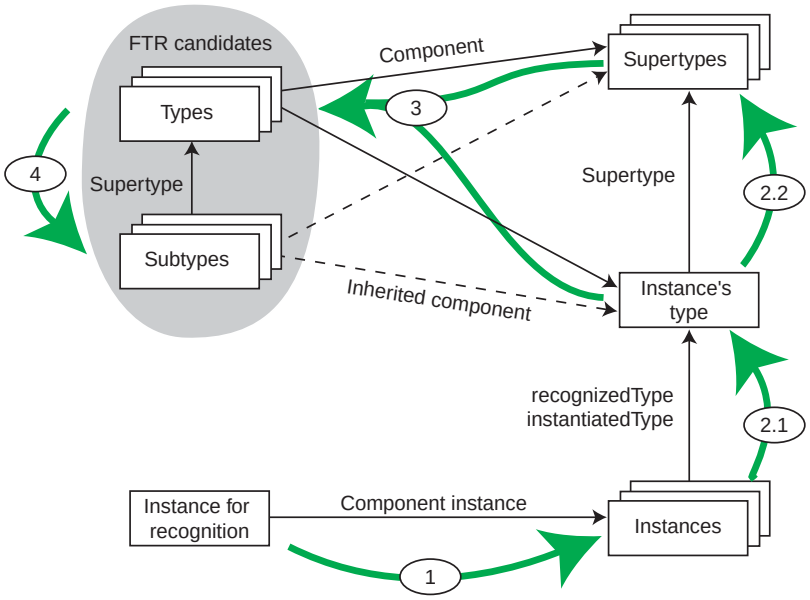

Figure 3 The algorithm for finding feature type candidates.
Legend: Rectangles represent feature types and instances. Thin straight arrows represent relations defined by FBM. Thick curvy (green if rendered in colour) arrows represent how the FTR process traverses the feature data.

Ad 2. The reason for including the supertypes is that it is always allowed to use a more specific feature instance for a component than is defined in the corresponding type. An example: When modelling a door, the ‘door’ type may define that a ‘door handle’ should be one of its parts. However, the designer may want to specify a specific make of door handle. Thus, the actual instance is more specific than the ‘door’ type prescribes. If we are about to apply FTR to the door instance, we need to consider the generic ‘door handle’ to be able to find the ‘door’ feature type.

The set comparison constraints have an impact on the on FTR process. The “subset” constraint expands a component to include other components. Thus, given the types in Figure 2, if we have an un-typed complex instance that has an instance of *BicycleWheel* as a *wheels* component, the FTR mechanism should suggest that the complex instance is recognised as a *Vehicle* type and as an incomplete *Bicycle* type. The former (Vehicle) because a *BicycleWheel* is a subtype of *Wheel*, which is defined as part of a Vehicle. The latter result (Bicycle) is obtained because a Bicycle has two *BicycleWheels* as *wheels* components, but is incomplete since it lacks the *frontWheel* and *rearWheel* components.

Selecting candidates

Currently, the (human) user does the selection from the list of found candidates. To aid the user, the list is ordered according to applicability. The ordering is based on several criteria, such as:

1. How well does the candidate’s set of components match the actual components of the instance?
2. How well do the candidate’s components match the instance’s component *fillers*?

In criterion 1, components are compared only if the *role types* are identical. Furthermore, the *role name*s need to be identical, or there need to be a constraint that equalises two role names (see above). There might be an exact match of components between the candidate type and the instance, or the type may have more or less components than the instance. If a type has components that the instance lacks, it represents a specialisation of the instance. Choosing such a type implies adding the missing components to the instance.

Criterion 2 might introduce an element of recursion, since correct types of the instance’s fillers need to be known before the evaluation can be made. Currently, however, no typing of the fillers is made. The main reason for this is that currently the user needs to make the final selection of the type. It would be too confusing to repetitively have to select the type of subcomponents. To be efficient, recursion needs automatic type selection.

The problem of how to automatically select types is studied in the second year of the research project (2002), but the subject is not ready for conclusions at the time of writing. Nevertheless, it can be said that the final outcome of an automatic FTR process depends on what weights are given to the different matching criteria, and the problem consists of determining what impact different weights will have. Plausibly, different settings would be preferable for different purposes. What is already clear is that an automatic FTR process should not add information to a model. Thus, an automatic process will not consider candidates that might be presented to a user as suggestions for further specifying a design model.

Conclusions and future work

The feature based modelling frameworks and the Design Knowledge Servers technology can provide the functionality needed for serving building product information.

Feature type recognition is a powerful technology for searching semantically high-level data, and is an important part of the DesKs technology. FTR makes it possible to find building products based on functional requirements of building parts.

The DesKs technology will be further developed at the Design Systems group at the Eindhoven University of Technology, as part of the VR-DIS research programme [de Vries et al, 2001].

Using a property-oriented design system involves adding properties to objects that represent the design and its parts. This allows the user to complete the definition of a part in one simple action by using FTR to find applicable types for the part and then to choose the best one, e.g. one that represents a product supplied by a manufacturer. Well-designed type libraries will ensure that the chosen product complies with the requirements as expressed in the model. To actually provide the well-designed feature libraries is a major task and beyond the scope of the current research project. It involves extensive analysis of the needs of all different actors in the design, construction, use, and management of buildings, and thus forms the object for future research.

FTR and the semantic web

Feature based modelling aims to heighten the level of semantic content of design modelling. Feature type recognition augments this capacity by introducing a kind of semantic analysis of feature models. With the introduction of DesKs servers, this capacity is extended to the Internet. The DesKs server approach

promises a network of interconnected design knowledge servers that will make semantically high-level design information available to any designer connected to the network.

Currently an effort is made to create the so-called semantic web [Berners-Lee et al. 2001]. The aim is to enable computer software to access the semantic content of data and to assist humans in finding meaningful information. Obviously, this work has much in common with the research on design knowledge servers and feature type recognition. It may thus be a task for the future to study how DesKs servers and FTR can fit into the general idea of the semantic web.

Acknowledgements

The reported research project is carried out by Sverker Fridqvist while holding a post-doc position at the Design Systems group of the Faculty of Architecture at the Eindhoven University of Technology.

References

Augenbroe, G. (1998) Building Product Information Technology, Executive white paper, Construction Research Center, Georgia Institute of Technology, 1998.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001) The Semantic Web, Scientific American, May 2001.

Fridqvist, S. (2000) Property-Oriented Information Systems for Design, prototypes for the BAS·CAAD System, PhD thesis, Lund University, Sweden.

Ekholm, A. and Fridqvist, S. (1996) Basic Object Structure for Computer Aided Modelling in Building Design in: Turk, Ž. (ed) *Proceedings of the CIB-W78 International Conference “Construction on the Information Highway”,* University of Ljubljana, Slovenia, pp 197-206.

Jain, S. and Augenbroe, G. (2000) The Role of Electronic Product Data Catalogues in Design Management, presented at the CIB W96 *Conference Design Management in the Architectural and Engineering Office*, May 19-20, 2000, Atlanta, Georgia, USA.

van Leeuwen, J.P. (1999). Modelling Architectural Design Information by Features, an approach to dynamic product modelling for application in architectural design, PhD thesis, Eindhoven University of Technology, the Netherlands.

van Leeuwen, J.P., Hendricx, A., and Fridqvist, S. (2001) Towards Dynamic Information Modelling in Architectural Design. In*: Proceedings of the CIB-W78 International Conference “IT in Construction in Africa” 2001*. CSIR, Division of Building and Construction Technology, pp 19.1-14.

van Leeuwen, J.P. and Fridqvist, S. (2002) On the Management of Sharing Design Knowledge, in: *Proceedings of the CIB W78 conference* “*Distributing Knowledge in Building”*, June 12-14 2002, Aarhus, Denmark.

de Vries, B, Achten, H, Coomans, M.K.D, Dijkstra, J, Fridqvist, S, Jessurun, J, van Leeuwen, J.P, Orzechowski, M, Saarloos, D, Segers, N, Tan, A. (2001). The VR-DIS Research Programme, Design Systems group. In*: Proceedings of the Computer Aided Architectural Design Futures Conference 2001*, 8-11 July 2001, Eindhoven University of Technology, The Netherlands, pp 795-808.