Technical Note on a
CONCEPT FOR THE XFEATURE TOOL

Written by A. Pasetti of P&P Software GmbH / ETH Zurich and
O. Rohlik of ETH-Zurich

Written By:	A. Pasetti (P&P Software GmbH / ETH-Zurich)
	O.Rohlik (ETH-Zurich)
Date:	15 June 2005
Issue:	1.3
Reference:	PP-TN-XFT-0001

Table of Contents

1 GLOSSARY AND ACRONYMS

The table defines the most important technical terms and abbreviations used in this document. The definitions given in the table are those used in the document which may not coincide with the definition for the same terms given in other documents.

Term	Short Definition
Application Meta-Model	An XML Schema that defines the language to express an application model expressed as XML document.
Application Model	A feature model expressed as an XML document that describes an instance of a family.
Application Meta-Model Generator	An XSL program that generates a *feature meta-model* representing the *application meta-model* from the *feature model* representing the *family model*.
Component	A unit of binary reuse that exposes one or more interfaces and that is seen by its clients only in terms of these interfaces.
Component-Based Framework	A software framework that has components as its building blocks.
Constraint Checker	An XSL program that is run on a *feature model* (expressed as an XML document) to verify whether it complies with all its *global constraints*.
Constraint Meta-Model	An XML Schema defining a language to express a *constraint model*.
Constraint Model	A description (as an XML document) of the *global constraints* applicable to a feature model.
Constraint Model Compiler	An XSL program that generates the *constraint checker* program from the *constraint model*.
Design Pattern	A description of an abstract design solution.
Domain Model	A description of the common and variable *features* of the applications in a *software product family* together with a definition of their semantics and a specification of the constraints on their combinations
DSL	Domain Specific Language (a language that is created to describe applications or components in a very narrow domain).
Family Meta-Model	An XML Schema that defines the language to express a family model expressed as XML document.
Family Model	A feature model describing a product family expressed as an XML document .
Feature	A characteristic of a system that is relevant to its users
Feature Diagram	A tree-like representation of a feature model
Feature Meta-Meta-Model	A Schema that defines the (XML Schema) language to express feature meta-models.
Feature Meta-Model	An XML Schema that defines the (XML-based) language to express the feature models.
Feature Model	A description of a set of features and their legal combinations.
Framework Domain	The set of applications that can be instantiated with the help of the framework
Framework Instantiation	The process whereby the reusable assets offered by the framework are used to construct an application within its domain
Generative Programming	A software engineering paradigm that promotes the automatic generation of an implementation from a set of specifications.

Global Constraint	In a feature model, a constraint applicable to a set of features that are not children of the same parent feature.
Local Constraint	In a feature model, a constraint applicable to a set of features that are children of the same parent feature.
Object Oriented Framework	A framework that uses inheritance and object composition as its chief adaptation mechanisms.
OBS	The On-Board Software.
Product Family	A set of applications that can be built from a pool of shared assets
Software Framework	A kind of *software product family* where the shared assets are software components embedded within an architecture optimized for a certain domain
Software Product Family	A set of software applications that can be built from a pool of shared software assets
Software System Family	A synonym for *software product family*

2 REFERENCES

[Afi04] Automated Framework Instantiation Project Web Site, http://www.pnp-software.com/AutomatedFrameworkInstantiation

[Ass05] http://www.assert-online.net/

[Cec04] Cechticky V, Pasetti, A, Rohlik O, Schaufelberger W, *XML-Based Family Modelling,* in: Bosch, J; Krueger, C; *Software Reuse: Methods, Techniques, and Tools,* LNCS Vol. 3107, Springer-Verlag, 2004

[Cza00] Czarnecki, K., Eisenecker, U.: Generative Programming – Methods, Tools, and Applications. Addison-Wesley, 2000

[Xfe04] XFeature Web Site: http://www.pnp-software.com/XFeature

[Pas02] Pasetti A (2002) Software Frameworks and Embedded Control Systems, LNCS Series, Springer-Verlag

[Pro02] Provost W, *Working with a Meta-Schema*, paper written in 2002 and available from: http://www.xml.com/pub/a/2002/10/02/metaschema.html

3 INTRODUCTION

This document describes and justifies the concept for the XFeature tool. XFeature is a feature modelling tool which supports the modelling of product families and of the applications instantiated from them. XFeature allows the users to define the feature meta-model as a tool plug-in. XFeature is provided as a plug-in for the Eclipse platform. The tool is available as free and open software (under a GNU General Public Licence) downloadable from the project web site. This web site also gives access to detailed design information about the tool and to its configuration files.

The initial development of the tool concept was done in the context of an ESA study (contract 18499/04/NL/MV) to develop a prototype feature modelling tool. The study was done by P&P Software GmbH as prime contractor with the Institute of Automatics of ETH-Zurich as subcontractor. Further extensions and use of the tool were done at ETH-Zurich in the context of the ASSERT project [Ass05].

The concepts presented in this document will often be illustrated through examples based on the feature modelling technique presented in [Cec04]. Readers may want to familiarize themselves with the content of [Cec04] before reading this document. Reference [Cec04] can be accessed from the XFeature web site. A summary of the feature meta-model presented in [Cec04] can also be found in section 12.1.

4 Overview of Feature Modelling

In general, a feature is a characteristic of a system that is relevant to its users. A feature model is a description of a set of features, usually the features of a set of related systems or of their component parts.

Feature models have recently emerged as one of the key supporting technologies for the development of software product families. An appreciation of the value of feature modelling therefore requires some understanding of the product family concept.

A *software product family* is a set of systems that can be constructed from a pool of shared software assets. The shared assets can be seen as generic *building blocks* from which systems in the family can be built. Usually, a product family is aimed at facilitating the instantiation of systems within a narrow domain. Figure 4-1 illustrates the concept of product family. On the left had-side, the building blocks offered by the product family are shown. These building blocks are used during the *family instantiation process* to construct a particular system within the family domain. The systems supported by a product family can either be hardware-based or they can be software applications. In the software world, the product family approach is currently the favoured technique to ensure a high level of software reuse and hence to contain software-related costs.

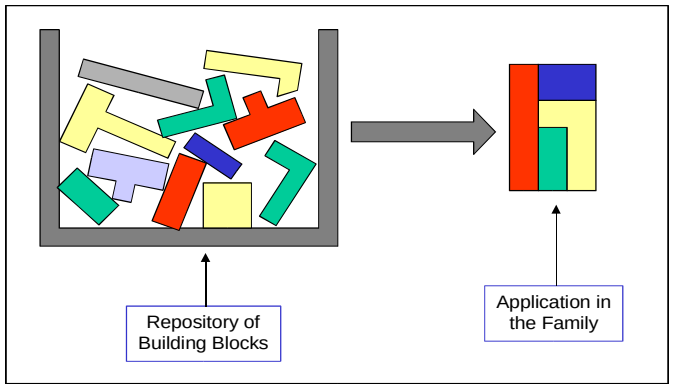

***Fig. 4-1**: The Product Family Concept*

The development process of a product family is usually broken into three phases (see figure 4-2). In the *domain analysis phase*, the set of systems that must be covered by the family is identified and characterized. In the *domain design phase*, the building blocks that populate the family repository are defined and designed. Finally, in the *domain implementation phase,* the building blocks are developed and implemented. Feature modelling can play a role in the domain analysis and in the domain implementation phases.

When a feature model is used to describe the domain of a product family, its purpose is to identify all the features that can potentially appear in the applications to be instantiated from the family. Additionally, the feature model must identify the constraints on the legal combinations of features (i.e. constrains on which combinations of features may appear in the same application). A feature model can thus be seen as the specification of the product family (the equivalent at family level of the user requirements at application level).

The reusable building blocks that are developed in the domain implementation phase are normally configurable. They must be configurable in order to be able to support the implementation of

multiple systems within the family domain. Each system to be instantiated from the family corresponds to one particular configuration of the building blocks in the family repository. A feature model can be used to represent the set of possible configurations of the family building blocks. In this case, a feature covers one configuration option for one particular building block and the feature model describes all possible configurations of the building blocks together with the constraints on their combinations.

An excellent general introduction to feature modelling and the role it plays in modern software engineering can be found in [Cza00].

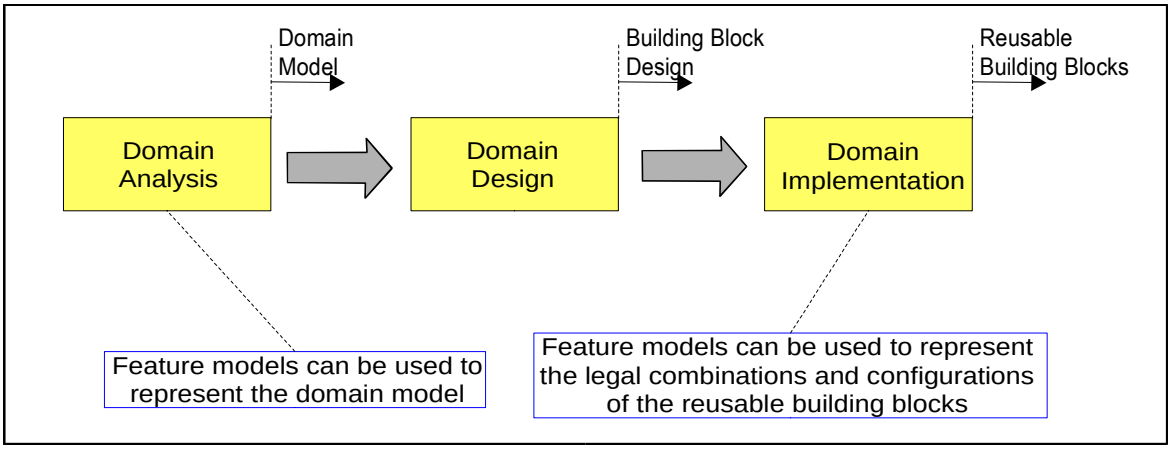

***Fig. 4-2**: The Product Family Development Process*

4.1 Representation of Feature Models

The most common representation of feature models is through FODA-style *feature diagrams*. A feature diagram is a tree-like structure where each node represents a feature and each feature may be described by a set of sub-features represented as children nodes. Various conventions have been evolved to distinguish between mandatory features (features that must appear in all applications instantiated from the family) and optional features (features that are present only in some family instances). Limited facilities are also normally available to express simple constraints on the legal combinations of features.

Figure 4.1-1 shows an example of a feature diagram for a family representing (much simplified) control systems. The diagram states that all control systems in the family have a single processor, which is characterized by its internal memory size, and have one to four sensors and one or more actuators. Sensors and actuators may have a self-test facility (optional feature). Sensors are either speed or position sensors whereas actuators can only be position actuators.

The figure also shows the most common convention for representing the cardinalities of features and the constraints on the combinations of sub-features that are children of the same feature. The feature diagram of figure 4.1-1 describes a family of control systems. Examples of systems that are within the family include:

- A control system with 3 position sensors, 2 position actuators, and a processor with 32 kBytes of memory

- A control system with 2 speed sensors, 3 position actuators, and a processor with 32 kBytes of memory. The sensor have a self-test capability.

Conversely, a control system with 5 sensors does not belong to the family because the family stipulates that there can be at most 4 sensors. A control system with a speed sensor and a position sensor does not belong to the family either because the family stipulates that control systems must have either position sensors or speed sensors, but not both.

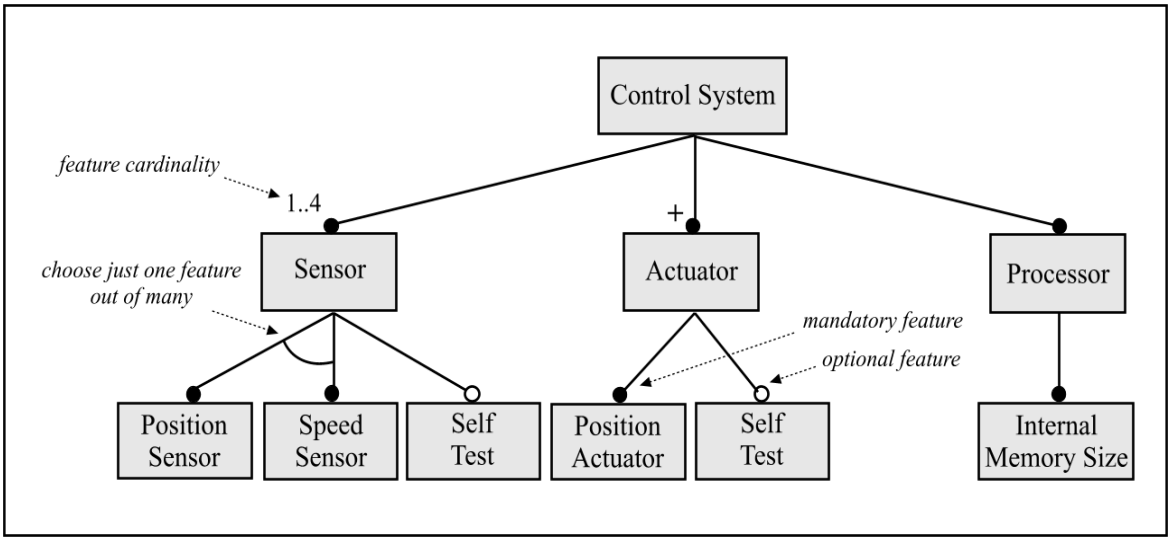

***Fig. 4.1-1**: Feature Diagram for a simplified Family of Control Systems*

Note that a feature model does not define the semantics of the individual features. This must be defined separately. Two basic approaches are possible. The first – and by far the most common – one is to define it in natural language, perhaps using some kind of domain dictionary. The second approach is to define the semantics "by translation", namely by mapping the features to some other entity with a well-defined semantics.

The figure above shows the visual representation of a feature diagram. Feature diagrams are usually serialized to an XML document. Given their tree-like character, an XML representation is a very natural means to represent feature diagrams.

4.2 Long-Term Goal

In order to put the development of the XFeature tool into perspective, it is useful to briefly describe the long-term vision that motivates the programme of work behind it. The objective is to develop a generic generative environment for automating the software product family instantiation process. The structure of such an environment is sketched in figure 4.2.

The generative environment consists of a family-independent skeleton. This skeleton is parameterized with a model of the family to be instantiated and with the family reusable software assets. The user provides as an input to the environment a specification of the target application (which must be within the domain of the product family). The environment reads this input and uses its knowledge of the family to automatically configure its reusable assets to create an application that implements the specifications expressed in the user inputs.

In the spirit of the XFeature project, the family model that parameterizes the environment is a feature model that describes the possible configurations of the family assets and the application specification is a model of an application within the family domain.

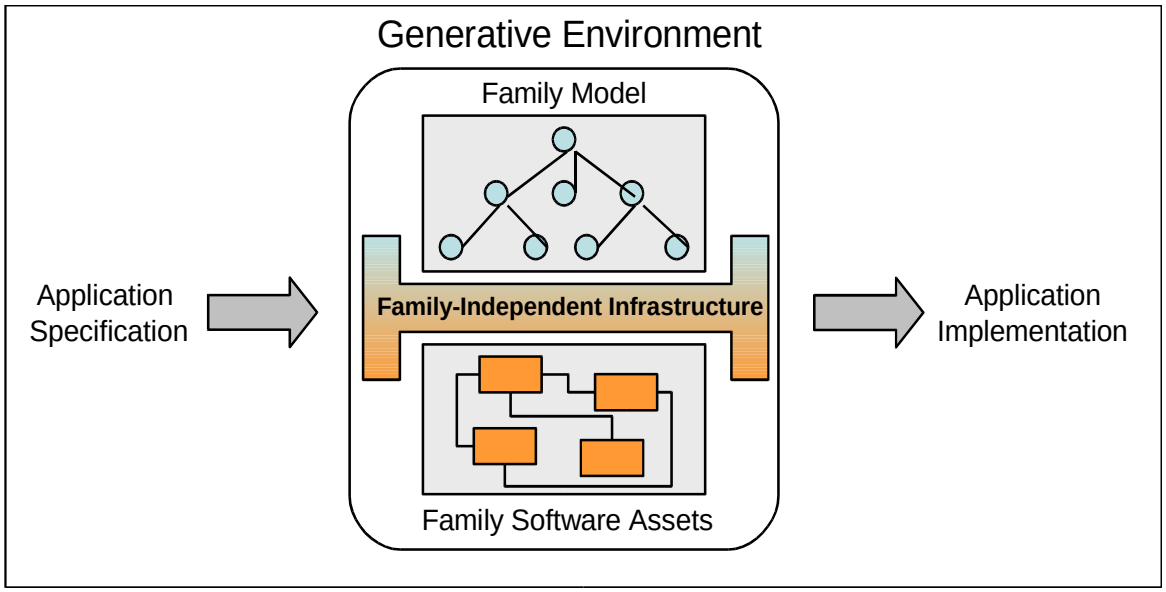

***Fig. 4.2-1**: Structure of Generative Environment for Software Product Families*

5 Feature Modelling Approach

This section describes the general feature modelling approach that is assumed by the XFeature tool. This approach is loosely based on ideas first put forward in [Cec04]. Since the proposed technique was primarily developed as an aid to the design and instantiation of software product families, it is described in terms of the family concept.

5.1 Modelling Levels

The proposed modelling approach recognizes three levels of modelling:

- Product Family Meta-Modelling Level
- Product Family Modelling Level
- Application Modelling Level

At *family meta-modelling level,* the facilities available to describe a product family are defined. The family meta-model is fixed and family-independent. At *family modelling level*, a particular product family is described. A family model must be an instance of the family meta-model. Finally, at *application modelling level,* a particular application is described. The application model serves as a specification of the application to be instantiated from the family. The application model must be an instance of the family model.

Both the family model and the application model are represented as feature models. The former describes the mandatory and optional features that may appear in applications instantiated from the family (together with the constraints on their combinations). The latter describes the actual features that appear in a particular application. The application model is seen as a feature model where all features are mandatory and where all variability has been removed.

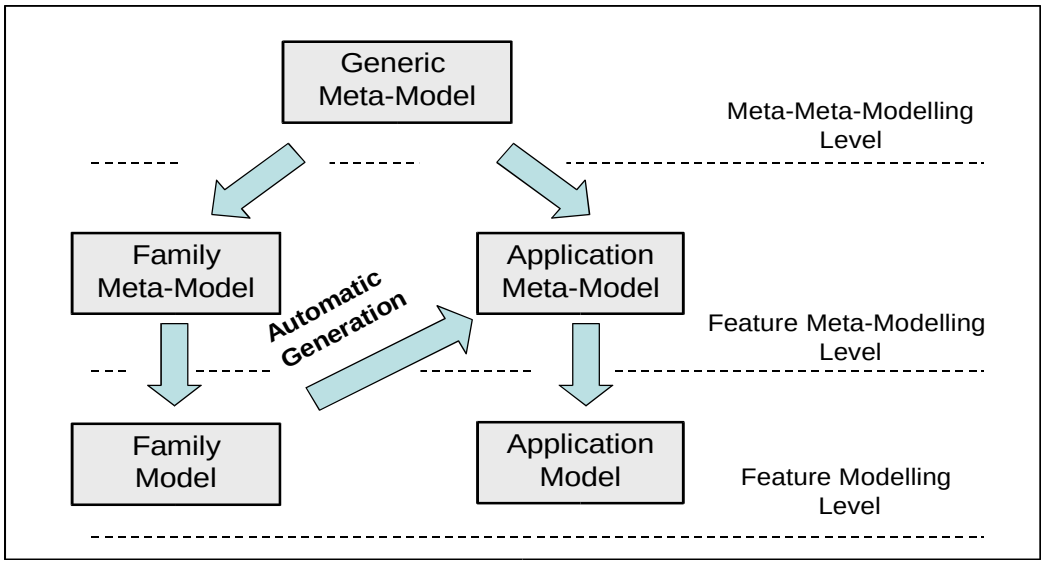

***Fig. 5.1-1**: Feature Modelling Architecture*

Since both the family and the application models are treated as feature models, it would in principle be possible to derive them both from a unique feature meta-model. However, the

characteristics of these two models are rather different and it was found that it is best to instantiate them from two distinct meta-models. This is illustrated in figure 5.1-1. The family model (a feature model) is instantiated from the family meta-model (a feature meta-model). The application meta-model (a feature meta-model) is automatically generated from the family model. The application models are instantiated from the application meta-model. Both the application meta-model and the family meta-model are instances of a single meta-meta-model.

With the concept of figure 5.1-1, a family is fully defined by its family model and by the program that generates the application meta-model from the family model.

5.2 Feature Model Syntax

Feature models require the definition of a concrete syntax to express them and the availability of a tool to support their definition. Several options are possible but the one that was selected in [Cec04] and that is adopted for the XFeature tool is XML-based. More precisely, feature models are expressed using XML-based languages and the relationship of instantiation between a model and its meta-model is expressed by saying that the XML-based model must be validated by the XML schema that represents its meta-model. The resulting modelling architecture is shown in figure 5.2-1. Both the family and the application models are expressed as XML-based feature models but they are instantiated from two different meta-models that take the form of XML schemas. The application meta-model is automatically generated from the family model by an XSL program (the *Application XSD Generator*). Note that, although not shown in figure 5.2-1, the family meta-model is itself constrained by a meta-meta-model (top-level box in figure 5.1-1).

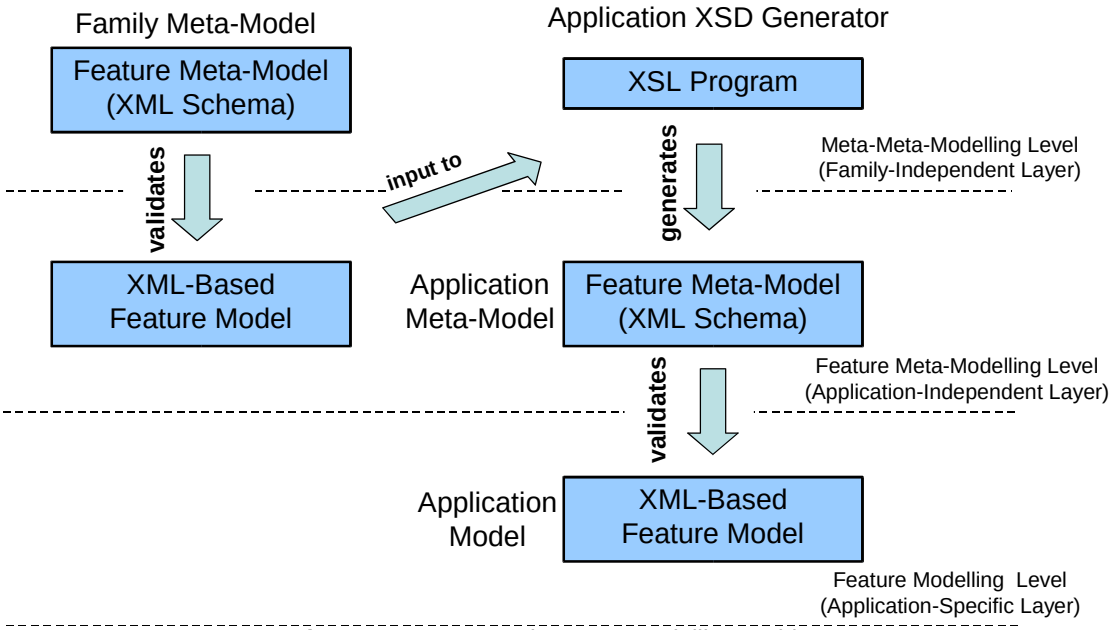

***Fig. 5.2-1**: XML-Based Feature Modelling Architecture*

With the architecture shown in figure 5.2-1, a family is completely defined by its feature model (an XML document) and by the Application XSD Generator (an XSL program).

Since an XML Schema defines an XML-based language, the application meta-model can also be seen as a the definition of a domain-specific language (DSL) for describing applications within the

family domain. This is relevant to the long-term plan of work described in section 4.2. The DSL induced by the family model can be seen as the language that should be used to specify the application to be instantiated with the generative environment. In other words, the environment is parameterized with a family model and this model is then transformed into an XML Schema that defines the language that should be used to express the specification of the target application to be instantiated from the product family.

5.3 Feature Composition Rules

A complete model of a product family consists of the list of all the features that may appear in applications instantiated from the family, together with a list of the *composition rules* that define the legal combinations of features that may appear in an application. A model of an application consists of a list of the features that appear in the application. The application represents an instantiation of the family if: (1) its features are a subset of the features defined by the family, and (2) its features comply with the composition rules defined at family level. Consequently, a complete feature modelling approach must offer the means to specify both features and their composition rules.

Two types of composition rules can be distinguished. *Local composition rules* express constraints on the combinations of sub-features that are children of the same feature. A local composition rule might, for instance, express a constraint that a certain feature can only have one sub-feature or that it can only have one sub-feature selected out of two possible options. *Global composition rules* express constraints on relationships between non-contiguous features in different parts of the feature diagram. A typical example of such rules are so-called "require-exclude relationships" (expressing the condition that the presence of a certain feature is incompatible with, or requires, the presence of another feature in a different part of the feature diagram).

At first sight, the approach sketched in the previous section cannot accommodate global composition rules. It uses XML documents to express both the family and the application models and it automatically derives an XML schema from the family model. The relationship of instantiation between the family model and the application model is then expressed by saying that the XML document representing the application model must be validated by this XML schema. This approach has the virtue of simplicity but is limited by the expressive power of an XML schema which allows only comparatively simple composition rules to be expressed. In practice, it is only *local composition rules* that can be easily expressed.

The use of an XML-language to express the application model, however, opens the way to more sophisticated approaches to expressing global composition rules. Arguably, the most obvious and the most flexible way to do so is to encode the general constraints as XSL programs that are run on the XML-based model of the application and produce an outcome of either "constraint satisfied" or "constraint not satisfied". Such an approach is powerful but has the drawback that it requires the family designer to be proficient in XSL.

In order to avoid this drawback while still exploiting the power of XSL to express general feature composition constraints, an alternative approach is proposed where the composition constraints are expressed in a dedicated language and a compiler is then provided to translate the constraint model expressed in this language into an XSL program that checks compliance with the constraints at application model level.

For consistency with the overall XML-based approach, the language to express the global constraints can be an XML-based language encoded as an XML Schema and the compiler for the

constraint model can be an XSL program. The following terminology is used in the rest of this document:

- The *Constraint Meta-Model* is the XML Schema that defines the XML-based language used to express the global constraints.
- The *Constraint Model Compiler* is the XSL program that reads the global constraints model (expressed as an XML document validated by the Constraint Meta-Model XML Schema) and transforms it into an XSL program that checks whether a certain feature model complies with certain global constraints.

6 XFeature Tool Approach

The previous section describes a general approach to feature modelling. The present section describes how this general modelling approach is translated into an approach for developing the XFeature tool.

6.1 Conceptual XFeature Structure

The purpose of the XFeature tool is to support family modelling activities by providing a GUI-based environment where users can define both the model of a family and the models of the applications within the family.

The XFeature model is based on feature modelling techniques in the sense that both the family model and the application models are expressed as feature models. The XFeature tool is based on the feature modelling approach described in the previous section. In particular, it sees a feature model as an instance of a feature meta-model.

In order to reduce its complexity, the XFeature tool does not differentiate between family and application models. The tool is only capable of supporting the definition of a feature model as an instance of a certain meta-model but it does not make any distinction between the situation where the feature model represents a family, and the situation where the feature model represents an application.

The XFeature tool, in other words, simply offers an environment where users can construct a feature model as an instance of a certain meta-model. The nature of the meta-model determines whether the feature model is a *family* model or an *application* model. This distinction, however, has no impact on the tool itself that is only concerned with feature models and ensuring that they comply with their respective meta-models.

One consequence of the above is that the tool should not enforce any particular feature meta-model. The feature meta-model is a user-defined parameter for the tool. This allows its users to use it to perform both family modelling (by parameterizing it with a feature meta-model that represents a family meta-model) and application modelling (by parameterizing it with a feature meta-model that represents an application meta-model).

It can be expected that different users will want to use different graphical notation to represent their feature models. In order to provide flexibility in this respect, the XFeature tool is also designed to be parameterized with a so-called *display model*. The display model defines how the individual elements of the feature model (the nodes of the feature diagram, the lines linking parent to children nodes, etc) should be rendered graphically.

Fig. 6.1-1: XFeature Tool Structure

The resulting conceptual structure of the XFeature tool is shown in figure 6.1-1. The figure indicates that the feature meta-model is expressed as an XSD file (an XML Schema, see section 5.2) and the display model is expressed as an XML file.

6.2 Relationship to Family Modelling

Given the structure described in the previous subsection, it would have been possible to restrict the XFeature tool to be a mere feature modelling tool and to completely decouple its expected primary usage as an aid in modelling product families. The relationship between the family and application models that the users wishes to build and the feature models that the tools allows them to build would then have been entirely implicit and hidden from the tool.

This approach would have been possible but was judged to be inconvenient for the target users of the tool. An intermediate approach was selected that preserves the independence of the GUI part of the tool from the distinction between family and application models but provides facilities to make the manipulation of feature models that represent family and application models easy.

The resulting structure of the tool is shown in figure 6.2-1.

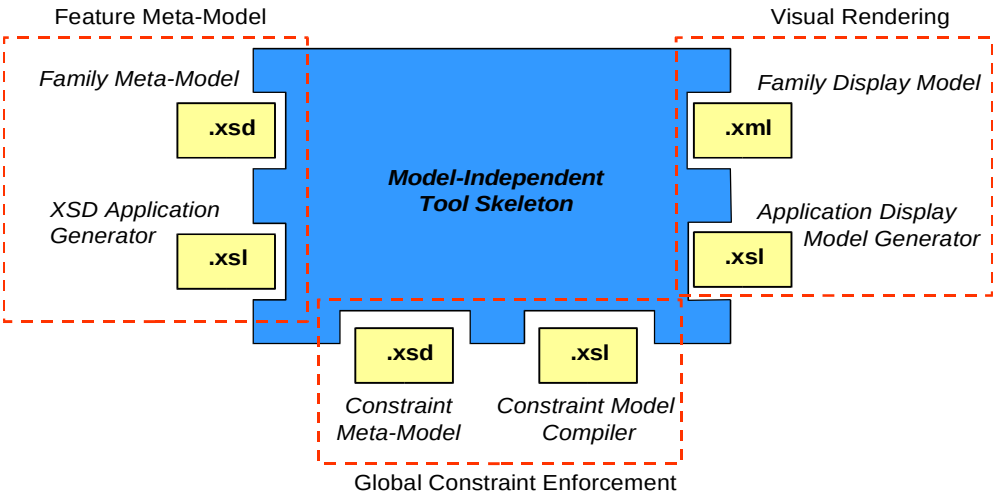

Fig. 6.2-1: XFeature Tool Structure

As indicated in the figure, the tool is parameterized with the following elements:

- A family meta-model (an XML Schema representing a feature meta-model)
- An application meta-model generator (an XSL program that translates a feature model representing a family model into an XML Schema representing the feature meta-model for the application model)
- A display model (an XML document that defines how each element in the family and application models should be displayed)
- An application display model generator (an XSL program that translates a display model for a family into the display model for the applications instantiated from the family)

The GUI part of the tool still only handles a feature model as an instance of a feature meta-model but the tool offers a shortcut to allow an application meta-model to be directly generated from the its family model.

For completeness the figure also shows two additional plug-ins for the tool that relate to its ability to enforce global composition rules on the feature models. These two plug-ins are:

- A global constraint meta-model (an XML Schema that defines an XML-based language to define global constraints)
- A global constraint model compiler (an XSL program that processes a constraint model and generates the global constraint checker XSL program)

In the present version of the tool, these two plug-ins however are not user-defined parameters. They are fixed and are provided with the tool. They are implemented as plug-ins in order to allow for future extensions where users may be given the option to define their own global composition rules.

6.3 Enforcement of Global Constraints

The tool does not directly enforce global composition constraints. However, it provides facilities for global constraints to be defined and for compliance with them to be checked by running the constraint checker. The approach is that described at the end of section 5.3. More specifically, the XFeature tool offers the following two files:

- A *global constraint meta-model* encoded as an XML Schema that defines an XML-based language to define global constraints
- A *global constraint model compiler* encoded as an XSL program that processes a global constraint model and generates the global constraint checker XSL program

Users define a *global constraint model* as an XML document that must be validated by the *global constraint meta-model*. This constraint model expresses the global constraints that apply to a particular feature model. The XFeature tool uses the *global constraint model compiler* to process the constraint model and to generate the *global constraint checker*. The latter is an XSL program that can be run on the target feature model to verify that it satisfies its global constraints.

The global constraint meta-model allows four types of global constraints to be expressed:

- *Requires Constraints:* constraints whereby the presence of a certain feature A in the feature model requires a second feature B to be present (i.e. A situation where A is present but B is missing represents a violation of the global constraints)

- *Exclude Constraints:* constraints whereby a certain set of features are mutually incompatible (i.e. If any two or more feature from the set of incompatible features are present, then the global constraint is violated)

- *Conditional Constraints::* constraints of the form: if (some feature-related condition holds) then (some feature constraint must hold). This type of constraints is a special form of the more general "custom constraint". It is provided as a separate type of constraint for convenience and because experience shows that it is often found in practice.

- *Custom Constraints*: arbitrary constraints expressed as XPath expressions

It is expected that most users will only need the first three types of constraints.

6.4 Provision of Default Models and Meta-Models

Although the tool is model-independent in the sense that it will not enforce any specific family meta-model, or display model, or constraint model, it provides a small number of sets of related default models. These are described in section 12.

7 High-Level XFeature Components

At any given time, the XFeature operates upon a feature model that can be edited by the user. This is called the *current model*.

The following high-level components contribute to the editing of the current model:

1. *Graphical User Interface (GUI) Component*: this component handles the interaction with the user and the construction and editing of the current model in a graphical environment.
2. *Feature Meta-Model (FMM) Component*: this component maintains the XSD representation of the meta-model for the current model.
3. *Feature Model (FM) Component*: this component maintains the XML representation of the current model.
4. *Display Model (DM) Component*: this component maintains the XML representation of the display model that is being used to display the current model.
5. *Constraint Model (CM) Component*: this component expresses the global constraints applicable to the current model.
6. *Constraint Checker (CC) Component*: this component is responsible for performing constraint checks upon the current model.

Additionally, the FMT has a further high-level component called the *command processor*. This component is responsible for executing non-GUI related commands issued by the user. Examples of such commands could be requests to generate an application meta-model from a given family model or requests to perform a constraint check on the current model.

The resulting high-level architecture of the FMT is sketched in figure 7.1. The following subsections describe each component in greater detail.

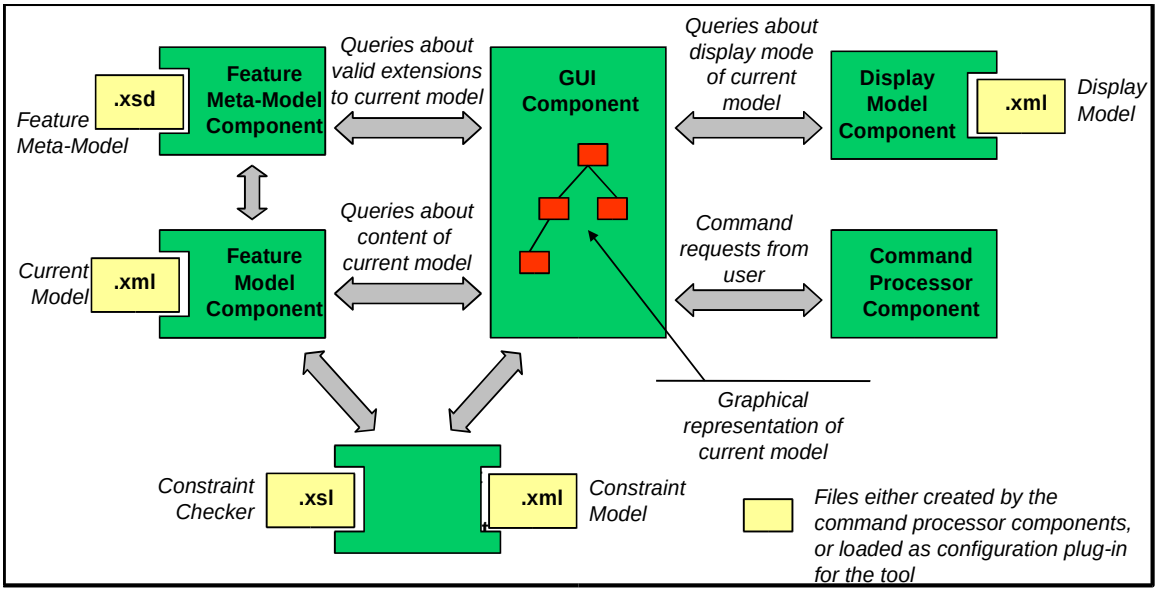

Fig. 7.1: High-Level Architecture of XFeature Tool

7.1 GUI Component

The GUI component is built as an Eclipse plug-in. It has two primary functions. The first one is to provide a command interface to the user allowing him to issue commands. The execution of the commands is, however, delegated to the command processor component. The interaction between the GUI and command processor components is based on the command design pattern.

The second function of the GUI component is to provide an editing interface allowing the user to edit a feature model. The GUI component queries the feature meta-model component to know at any given time what are the legal extensions to the current model. It then uses this information to offer to the user (the person who is editing the feature model) a range of options for the legal extension of the feature model.

Whenever a change to the current model is made, the GUI component fires an event to inform registered observers (typically, the feature model component) of the change.

7.2 Feature Meta-Model Component

The feature meta-model component is responsible for parsing the XML Schema representing the feature meta-model and for providing information to the GUI component about the legal extensions of the current feature model.

The exchange of information between the GUI component and the feature meta-model is as follows: (1) the GUI component tells the feature meta-model component which node in the current feature model is currently being edited, and (2) the feature meta-model component tells the GUI component what are the legal extensions of the feature model at that node.

7.3 Feature Model Component

The feature model is stored as an XML document. This component uses the DOM representation of the XML document. The component is therefore implemented as basically a wrapper for the DOM model.

The feature model must be synchronized with the model update operations performed by the user in the GUI environment. This component therefore acts as an *observer* (in the sense of the observer design pattern) for events fired by the GUI component to signal an update to the feature model.

In order to avoid the need for implementing a layouting algorithm, the feature model component also stores information about the layout of the current model. Thus, when a model is saved and then re-opened, its layout at the time it is re-opened is the same as it was at the time it was closed.

7.4 Constraint Checker and Constraint Model Components

This component runs the constraint checker upon the current model as it is stored in the feature model component. The request to run the constraint checker might come from the GUI component (further to a request by the user). Alternatively, the constraint checker component might perform checks run on a periodic basis (but on a low-priority thread to avoid interference with other parts of the application).

The outcome of the check is passed to the GUI component that can use it to display it in textual form or to highlight offending elements in the current model.

The constraint checker is an XSL program that is automatically generated by processing the constraint model, which expresses the user-defined global constraints applicable to the current model.

7.5 Display Model Component

This component answers queries from the GUI component about how a certain model element should be displayed. The answer to the query is found by processing the display model that is stored as an XML document.

7.6 Command Processor Component

This component executes command requests that it receives from the GUI component. This component is implemented as a command handler in the sense of the command design pattern.

8 Meta-Meta-Model – Conceptual Structure

The XFeature tool allows users to define their own feature meta-model. The feature meta-model is expressed as an XSD document (an XML Schema). However, not all XSD documents represent valid feature meta-models. In order to formalize the notion of "valid feature meta-model" and to allow a clear demarcation between XSD documents that can serve as feature meta-models (and can therefore be loaded into XFeature) and other XSD documents, the notion of feature meta-meta-model is introduced. Valid feature meta-models must be instances of a feature meta-meta-model.

In terms of the representation in figure 5.1-1, the feature meta-meta-model is the box at the top of the figure. This section describes the conceptual structure of the feature meta-meta-model. The next section considers its implementation aspects.

Conceptually, the feature meta-meta-model enforces a certain structure for the feature models. This structure can itself be represented as a feature diagram that is shown in figure 8.1. As shown in the figure, a feature model has a tree-like structure. The feature model consists of a tree of *nodes*, each node may have a set of *properties* attached to it, and the properties are divided into sets of *property sets*.

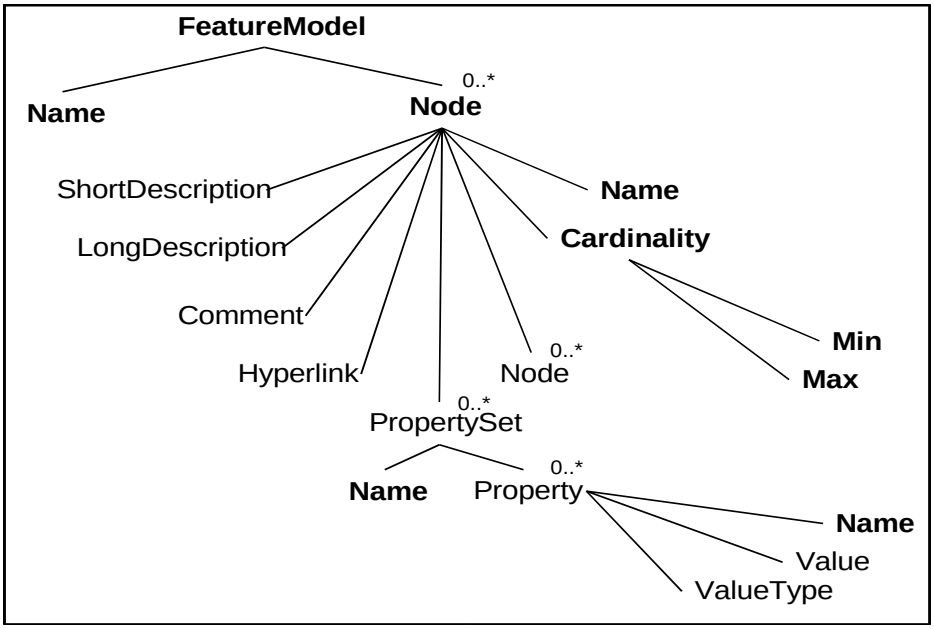

Fig. 8.1: Conceptual Structure of a Feature Model

At the top level, a feature model has a *name* and one *node* (the *root node*). Each node is characterized by the following attributes:

- A mandatory *name* that uniquely identifies the type of the node
- An optional *short description* (one sentence) that describes the node. This can typically be used as the text in a floating box over the graphical representation of the node.
- An optional *long description* that describes the node in greater detail. This can be used to automatically generate a textual documentation of a feature model.

- An optional *hyperlink* that can be used to create links to other information sources related to the node.
- An optional *comment* that can be used to insert textual comments that are not intended for display.
- A mandatory *cardinality* that defines the minimum and maximum cardinality of the node. Note that the semantic of the cardinality information is not defined within the GUI component.
- Zero or more *property sets* that defines a set of logically related *properties.* A property in turn defines a typed value that can be associated to the node.
- A set of zero or more *children nodes*.

The feature meta-meta-model enforces the structure defined above. This structure is hardcoded into the XFeature tool and is used by the tool to decide how to represent a feature model.

A user-defined feature meta-model defines the type of nodes that can appear in a feature model, the restrictions on their parent-child relationships (i.e. which node types can be children of which node types), and the *properties* associated to each node. For instance, in the case of the ICSR family meta-model described in section 12.1.1 and in reference [Cec04], the following types of nodes are foreseen:

1 Nodes representing a *feature*
2 Nodes representing a *group*
3 Nodes representing a *macro call*
4 Nodes representing a *feature macro with no extension*
5 Nodes representing a *feature macro with group extension*
6 Nodes representing a *feature macro with feature extension*

This meta-model additionally places some restrictions on the parent-child relationship between these nodes. Thus, for instance, a feature node cannot have other feature nodes as children. It can only have group nodes as children. Similarly, a group node cannot have other group nodes as children but it can have either feature or any type of macro nodes as children.

It is usually desirable to associate some values to a node in a feature model. The simplest option for doing so is to define a fixed structure whereby to each node is associated one – or perhaps a small number of – properties. This approach however is too rigid both because different meta-models have different needs and because, within the same meta-model, it is usually impossible to associate the same set of properties to each node or even to each type of node.

The next simplest approach is to allow the meta-model to specify which set of properties should be associated to each type of node. In the previous example, for instance, one could associate one set of properties to nodes of type *feature,* a second set of properties to nodes of type *group*, and so forth for the other node types. This approach is still too rigid as it does not allow different sets of properties to be associated to the same kind of nodes.

The approach adopted for XFeature assumes that all properties required within a model are gathered together in property sets and the meta-model specifies the property set that may be associated to each type of node. During the model definition phase, the user selects the property sets that are to be actually associated to each individual node.

9 Meta-Meta-Model – Implementation

The previous section presented the feature meta-meta-model at concept level. The present section discusses how this concept is actually implemented in the XFeature tool.

The purpose of the feature meta-meta-model is to encode the constraints on a feature meta-model that must be satisfied if the feature meta-model is to be handled and understood by the XFeature tool. There are two basic options for implementing such a meta-meta-model.

The simpler option is to hardcode the checks on the compliance of the feature meta-models with the XFeature-specific constraints in the XFeature tool itself. In practice, this could be done by using a simple XSL script to perform the checks on each newly loaded feature meta-model. This approach would work but would make it difficult to formally define the constraints to be satisfied by the feature meta-models and to ensure that these constraints are consistent with each other.

The second option, and the one that was adopted for the XFeature tool, is to treat the feature meta-model as an XML document that must be validated against a schema and to encode the XFeature-specific constraints in the schema. The validating schema becomes the feature meta-meta-model.

The feature meta-model is itself an XML Schema. It is therefore validated by default against the `XMLSchema.xsd` schema which is defined by the W3 consortium and which defines the XSD language. The problem of creating a feature meta-meta-model can therefore be seen as the problem of creating a schema that incorporates the standard `XMLSchema.xsd` schema and adds to it the constraints imposed by the XFeature environment. This is illustrated in figure 9-1.

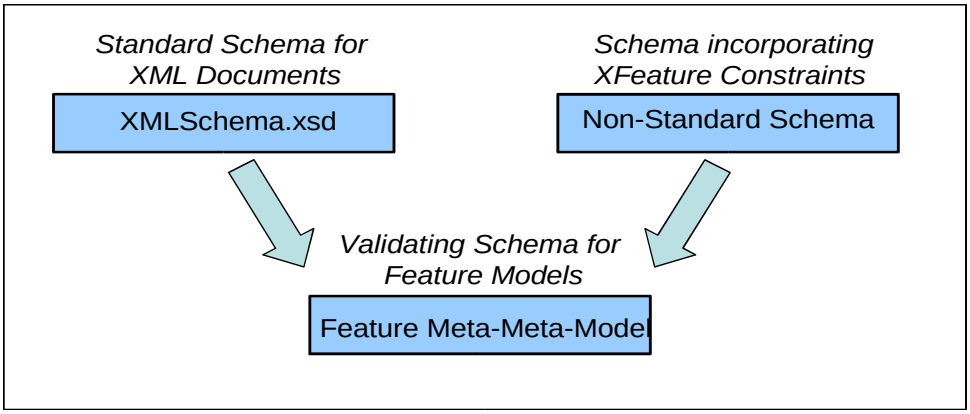

Fig. 9.1: Content of Feature Meta-Meta-Model

The feature meta-meta-model must capture the conceptual structure for feature models described in section 8 (see in particular figure 8-1). According to this structure, a feature model consists of a tree of *nodes* of which one is the *root node* and the others are children of either the root node or of other nodes. Each node may have a set of *properties* attached to it and the properties are divided into sets of *property sets*. The nodes are characterized by a cardinality (which in turn consists of a minimum and maximum cardinality), by a *name*, and by a number of optional attributes (comment, short and long description, and hyperlink).

A first simplification can be obtained by noting that the optional attributes can be seen as properties in the same property set. The feature meta-meta-model therefore pre-defines a property

set called `DocumentationPropertySet` that holds the comment, short and long description, and hyperlink properties. These attributes therefore do not need to be considered further[1].

The feature meta-meta-model assumes that a feature meta-model is encoded as an XML Schema and that each XML Schema element describes a feature model element. The possible feature model elements are: *root node*, *node*, *cardinality*, *property set*, and *property*. These feature elements are the elementary "building blocks" that the XFeature tool uses to build the visual representation of a feature model. The *name* of a node is the name of the XML Schema element that describes it[2].

The feature meta-meta-model must enforce three kinds of constraints on an XML Schema for that schema to be a valid representation of a feature meta-model:

- The display type constraint
- The structural constraint
- The syntactical constraint

These three types of constraints are described in greater detail in the next three subsections. The following subsection (subsection 9.4) describes how all these constraints are brought together in a single validating schema for XML Schemas that are intended to represent feature meta-models.

9.1 The Display Type Constraint

As indicated above, the feature meta-meta-model assumes that each feature element is described by an XML Schema element. The possible feature elements are nodes, their cardinalities, their property sets, and the properties in a property set.

In the XFeature meta-meta-model, this link between XML Schema elements and the feature elements is enforced by stipulating that all XML Schema elements should have a mandatory attribute representing the feature element which that schema element describes[3]. The name of this attribute is `displayType`. The feature meta-meta-model allows this attribute to have one of the following possible values:

- `none`: this value indicates that the element is not displayed by the XFeature tool
- `cardinality`: this value indicates that the element describes the cardinality of a node
- `node`: this value indicates that the element describes a node.
- `property`: this value indicates that the element describes a property of a feature

[1]The `DocumentationPropertySet` however is treated in a "special" way by the XFeature tool because the tool uses the information it provides to decorate the visual icons of the feature model. For the rest, however, this property set is treated like all other user-defined property sets.

[2]Thus, for instance, in the case of the ICSR feature meta-model (see section 12.1.3 and reference [Cec04]), six XML elements are defined whose names are: Group, Feature, Macro Call, Macro with No Extension, Macro with Group Extension, and Macro with Feature Extension (see also example in section 8). Each of these elements describes one of the node types that are possible in the ICSR feature models.

[3]In reality, for technical reasons that are discussed in the XFeature web site, the `displayType` attribute is implemented as a mandatory attribute for all *complexType* elements in an XML Schema representing a feature meta-model. In practice, this poses no problems because the feature meta-model elements are always defined as complex types.

- `propertySet`: this value indicates that the element describes property set, namely a container for a set of properties for the same feature
- `root`: this value indicates that the element describes the root node in a feature model

The `displayType` attribute defines how the XFeature tool should handle a certain element in a feature model. The tool handles feature models essentially by displaying them. Hence, the information in the `displayType` attribute is used by the tool to decide how a certain feature element should be displayed (as a node to be displayed in the tree editing are of the tool; as property to be displayed in the property sheet part of the tool; as cardinality information; etc). This is the reason why this additional information is called "displayType".

9.2 The Structural Constraints

The display type information discussed in the previous subsection allows the elements in the XML Schema to be mapped to the feature elements identified in figure 8-1. The structural constraints defined in this section instead enforce the mutual relationships between these elements. More specifically, the following constraints are enforced:

- element of `displayType` "`node`" can only contain elements of `displayType` "`node`", "`propertySet`", and "`cardinality`"
- element of `displayType` "propertySet" can only contain elements of `displayType` "`property`"

The first constraint ensures that a feature model is built as a tree of nodes whereas the second constraint enforces the fact that properties are contained in property sets.

9.3 The Syntactical Constraints

Feature meta-models are implemented as XML Schemas written in the XSD language. The meta-models are parsed and interpreted by the XFeature tool at run-time to decide what are the possible legal extensions of the feature model that is being currently edited. The XSD language can be very complex. In order to simplify the task of the XFeature tool, the XSD language is restricted by imposing certain syntactical constraints on the definition of the XML Schema that represents a feature meta-model. The following specific constraints are imposed:

- *Local* `complexType` *Definition Constraint*: inheritance of `complexType` in the feature meta-model is not supported. This is achieved by forbidding global `complexType` definitions. As a result, a feature meta-model can only consist of global element definitions where each element is either of simple type or contains local `complexType` definitions.
- *Simple* `complexType` *Structure Constraint*: it is not allowed to embed another `complexType` definition inside a `complexType` definition. If a `complexType` element contains elements that are of `complexType` then these elements may only contain references to globally defined elements.

9.4 Implementation Approach

The feature meta-meta-model is implemented as an XML-based validating schema that implements all the constraints imposed by the standard `XMLSchema.xsd` schema and in addition

implements the XFeature-specific constraints described in the previous three subsections. The implementation of these additional constraints is done using the following two approaches:

- Modification of the rules in the `XMLSchema.xsd` schema
- Embedding within the `XMLSchema.xsd` schema of additional rules based on the Schematron schema language

In order to limit complexity, the modification of the `XMLSchema.xsd` schema is done in a very limited manner using the "redefine" mechanism[4]. Essentially, this mechanism is used to enforce a constraint that each XML Schema complex element should have a mandatory attribute called `displayType` taking one out of a finite range of possible values. This thus enforces the first of the three XFeature-specific constraints (see section 9.1).

A second mechanism is required to encode all XFeature-specific constraints because the XML Schema is not powerful enough to express the structural and syntactical constraint defined in sections 9.2 and 9.3. A more powerful schema language is required. The one that was selected is Schematron.

The Schematron schema language differs from most other XML schema languages in that it is a rule-based language that uses XPath expressions instead of grammars. This means that instead of creating a grammar for an XML document a Schematron schema will make assertions applied to a specific context within the document. If the assertion fails, a diagnostic message that is supplied by the author of the schema can be displayed.

Schematron has a further advantage in that it can be embedded into an XML Schema document. As a result, it was possible to combine the use of the redefine mechanism to specify the display type constraints and the Schematron checks to specify the structural and syntactical constraints on the XFeature meta-models in a single schema document. This single schema document is the *XFeature meta-meta-model*.

The XFeature tool provides scripts to run the meta-meta-model upon an XML Schema to automatically check whether the schema complies with the XFeature-specific constraints. If this check is passed, then the schema can be loaded into XFeature which will be able to treat it as a valid feature meta-model.

Since the XFeature meta-meta-model is implemented as an XML Schema, it can also be used in a standard XML editing tool to provide code completion facilities that can help the user to construct his own feature meta-model.

[4] A technical description of this approach can be found in [Pro02].

10 Display Model Structure[5]

The display model defines how the elements in a feature model are displayed by the GUI component. Display information is provided for the nodes, for the property sets, and for the properties in each property set.

The display model is encoded as a user-defined XML document that must comply with a tool-defined XML schema.

The display model assumes a conceptual layout for the FMT editing interface as shown in figure 10.1. The editing area is divided into three parts. The *node editing area* is where the user can add or delete nodes to the feature model. The *property editing area* is where the user can edit the values of the properties (arranged in groups of property sets) of the node that has the editing focus. The *node tree display area* is a display-only (i.e. no editing) area where the user can see the tree structure of the feature model. This area will typically be used to quickly navigate the feature tree. Note that in practice the editing interface will also include other areas (for instance command bars, areas for managing projects and files, etc) but these are not relevant for the discussion of this section.

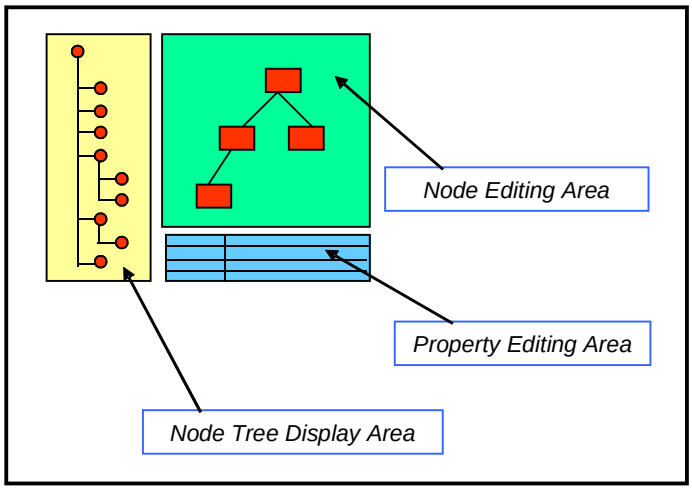

Fig. 10.1: Conceptual Structure of XFeature User Interface

The display model defines the appearance of the items in the editing areas shown in figure10.1. In general, only the part of the appearance of each item that cannot be modified by the user is covered by the display model. Thus, for instance, users can modify the relative position of some nodes in a feature tree in the node editing area. Hence, the position of a node in the display area is not defined in the display model. Conversely, the colour of the graphical icon representing a node is fixed and cannot be changed by the user. This colour is therefore defined by the display model.

The approach described in this section assumes that the display model is kept separate from the feature model. In an alternative approach, a single model is used that contains both the information about the features and their properties and the information about how they should be

[5] The discussion in this section – as in the rest of this document – is only related to the XFeature tool *concept*. Hence, the display examples are not necessarily representative from a visual point of view of the actual capabilities of the XFeature tool. Their intention is merely to illustrate the approach that the tool takes to encoding the information that defines the visual rendering of feature models.

displayed. This second approach would allow finer control over how each node is displayed: the display model defines how each *type of node* is displayed but a mixed model could defined how each *individual node* is displayed. On the other hand, the display information would "corrupt" the feature model and would make it heavier. This was the reason why an approach based on the separation of display information from the feature mode was adopted for the XFeature tool.

The next two subsections illustrate through some examples the kind of display attributes that are encoded in the display model.

10.1 Node Display

The GUI component treats a feature model as a tree where the display format for every kind of node is specified by the display model. The current version of the display meta-model distinguishes between two types of representations for a node: either as a box-shaped icon or as a dot-shaped icon.

As an example of how these two types of node icons could be used, consider the case of the ICSR family meta-model of section 12.1.1 and of reference [Cec04]. In that case, box-shaped icons would typically be associated to nodes representing "features" and dot-shaped icons would be associated to nodes representing "groups".

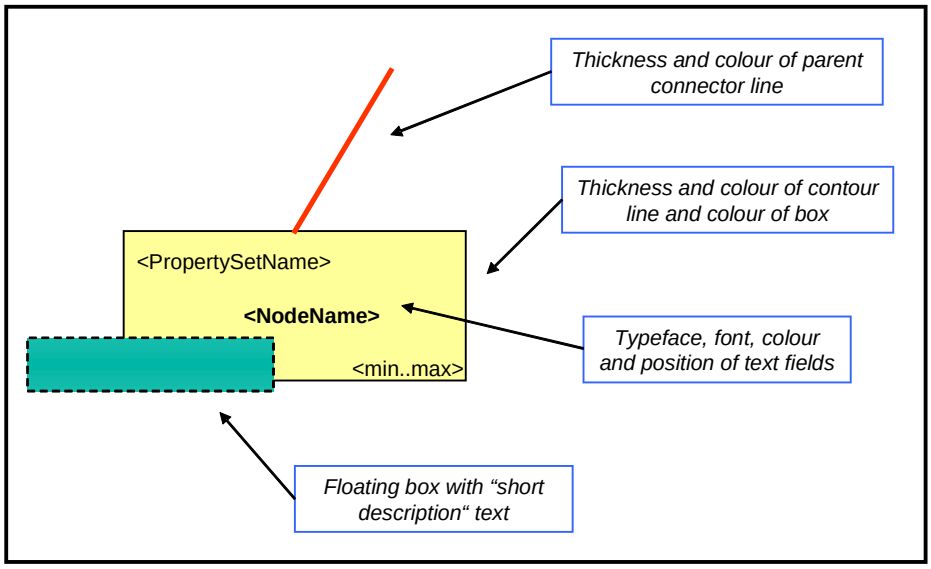

Fig. 10.1-1: Box-shaped icon example with indication of elements that are specified by the display model

The display model further allows users to specify the graphical attributes of both a box-shaped and a dot-shaped icon. Figure 10.1-1 considers the case of box-shaped icons. Additionally, the display model allows users to specify the type of connection between a node and its parent. Three possibilities are envisaged:

- The parent node may not exist. This would be the case of a root node such as, in the case of the ICSR meta-model, a feature macro node.
- The node is linked to its parent through a line whose characteristics (colour, thickness, style) are also specified in the display model.

- The node is directly attached to its parent.

Figure 10.1-2 shows an example of how the family meta-model of the ICSR paper might be represented (see also section 12.1.3). Nodes of type "feature" are mapped to box-shaped icons whereas nodes of type "group" are mapped to dot-shaped icons. Group nodes are specified to be attached to their parent nodes. The conventions that are used in the figure are different from those commonly used in feature diagrams (in particular with respect to group cardinalities) but the information content is the same. The important point to stress is that the diagram of figure10.1-.2 is obtained as a special case of a more general diagram and that the tool need not be aware of the semantical distinction between feature nodes and group nodes. All the tool needs to know is that there are two kinds of nodes – feature nodes and group nodes – with two different display conventions encoded in the display model.

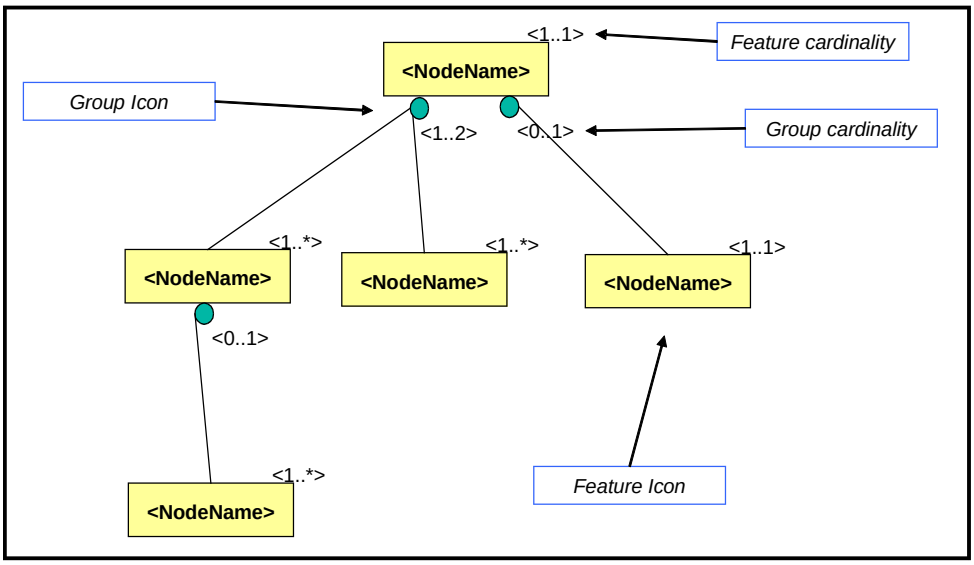

Fig. 10.1-2: Feature diagram example for the ICSR family meta-model

10.2 Property and Property Set Display

To each node a set of *properties* – organized in *property sets* – is associated. When the editing focus is on a particular node, any properties that are attached to that node are displayed in the property editing area. A typical display format is shown in figure 10.2-1. The display model allows the colour and font of all the text to be specified.

Note that the set of properties that are associated to a node may not be fixed. In the case of the ICSR meta-model of section 12.1.1 and reference [Cec04], for instance, each feature has a type. The set of properties attached to a feature could then be dependent on the feature type. In other words, each feature type has a different set of properties attached to it. This relationship between the type of a feature and the set of properties attached to it is encoded in the feature meta-model. During the feature model editing process, the user specifies the type of a feature and, depending on his choice, the set of properties to be edited changes. The XFeature tool uses the information in the feature meta-model to know which sets of properties to display in the property editing area when a particular feature has the editing focus.

<PropertySet_1 Name>	
<Property_1 Name**>**	**<**Property_1 Value>
<Property_2 Name**>**	**<**Property_2 Value>
<Property_3 Name**>**	**<**Property_3 Value>
<PropertySet_2 Name>	
<Property_4 Name**>**	**<**Property_4 Value>
<Property_5 Name**>**	**<**Property_5 Value>
<Property_6 Name**>**	**<**Property_6 Value>

Fig. 10.2-1: Example of property editing area

10.3 Future Extensions

The display meta-model is at present intended to define a *static* mapping from nodes and properties to graphical elements. The mapping is static in the sense that for each node type or for each property (as identified by its unique name), a fixed graphical element is associated. Greater flexibility could be provided by allowing the user to specify a *conditional mapping*, where the graphical elements are a function of the context within which nodes and properties are used. Thus, for instance, it could be possible to define a feature node to have a certain colour if the feature cardinality is 1 and to have a different colour if its cardinality is different. Similarly, it would be possible to give different appearances to the icons representing groups that have a minimal cardinality of zero or where the minimal and maximal cardinality are identical. This type of conditions could be best formulated using the XPath syntax.

The use of conditional mappings will be considered in future releases of the XFeature tool.

11 Other Tool Issues

This section briefly considers some secondary issues related to the implementation of the XFeature tool. Some possible future extensions are also considered.

11.1 Manipulation of Large Feature Models

The XFeature tool implements mechanisms for facilitating the manipulation of large feature models. Two such mechanisms are foreseen.

The *tree navigation mechanism* allows users to navigate the feature model in the node tree display area (see section 10). Users select items in the node tree and their selection causes the editing focus in the node display area to be changed to reflect the selection.

In future releases, users will also have the possibility to edit a feature model by operating on the node tree display.

The *sub-tree collapse mechanism* allows users to collapse a sub-tree of a feature model into a single node. The node represents the collapsed sub-tree.

The *modularization mechanism* will be introduced in future releases of the tool. It will allow users to break up a feature model into sub-models – the modules – that can be edited separately and are then referenced from within parent tree.

Note that the modularization mechanism should not be confused with the macro feature mechanism of the ICSR family meta-model of section 12.1.1 (see also [Cec04]). The latter is introduced by one particular meta-model whereas the former is defined at the level of the tool and should be applicable to any feature model.

11.2 Save Model Structure

A user constructs a feature diagram in the FMT editor. The *display model* defines the appearance of the items manipulated by the user in the editor. As the user defines new items in the FMT editor, the FMT internally constructs a *feature model*. The information stored in the feature model and in the display model however is insufficient to save the state of an editing session. Neither of the two models stores information such as the position of the nodes in the editing area. In order to allow a user to interrupt and resume an editing session, it is therefore necessary to introduce a further type of model: the *save model*. This model stores all the information required to save the current state of an editing session. For consistency, this model is encoded in XML however its precise format needs not be discussed here because the save model is an internal FMT model.

11.3 Tool Operational Modes

The feature modelling tool operates in two modes: *transformation mode* (TM) and *editing mode* (EM).

In transformation mode, the tool processes an existing feature model and generates the respective meta-model using the application XSD generator (see section 6). This operational mode would typically be used to perform the transition from the modelling at family level to the modelling at application level.

In editing mode, the tool lets a user create and edit a feature model.

The XFeature tool is intended to be configurable by the user. The behaviour of the tool is thus defined by its *configuration*. In transformation mode, the tool configuration is defined by the following files (see section 6):

1 The *Application Meta-model Generator*

2 The *Application Display Model Generator*

In editing mode, the tool configuration is defined by the following files:

- The *feature meta-model*
- The *display model*
- The *constraint model*
- The *constraint checker*

12 Default Tool Configurations

The XFeature tool is a configurable tool that allows users to define their own meta-models and display models (see section 6.2). The tool configuration is defined by a set of four files representing:

- The family meta-model (an XML Schema representing a feature meta-model)
- The application meta-model generator (an XSL program that translates a feature model representing a family model into an XML Schema representing the feature meta-model for the application model)
- The display model (an XML document that defines how each element in the family and application models should be displayed)
- The application display model generator (an XSL program that translates a display model for a family into the display model for the applications instantiated from the family)

The tool is delivered with two sets of configuration files representing two different feature meta-models. These default configuration files are useful in themselves but are also intended as a basis for the definition of new meta-models by users.

The two default configurations are referred to as *ICSR Configuration* and *Simplified ICSR Configuration*. The ICSR Configuration is so named because it is based on the family meta-model first introduced in a paper presented at the ICSR'04 conference [Cec04]. The Simplified ICSR Configuration is so named because it implements a simplified version of the ICSR family meta-model. This simplified configuration is also arguably the simplest kind of meaningful family meta-model.

The ICSR configuration is considerably more complex than the basic configuration. It therefore serves to illustrate the expressiveness of the XSD-based meta-modelling concept adopted by the XFeature tool.

The two default configurations are described in the next two subsections. The configuration files can be downloaded from the XFeature web site [Xfe05].

12.1 The ICSR Default Configuration

The description of this default configuration is split into four subsections corresponding to the four files that define a tool configuration.

12.1.1 Family Meta-Model

The ICSR default configuration is based on the ICSR family meta-model [Cec04]. This meta-model is shown in figure 12.1.1-1. Its description is divided into three parts covering the three groups of items enclosed in the red dashed contours. The items in the contour in the top right-hand corner are described first.

The basic ideas behind the ICSR meta-model can be summarized as follows. A feature can have sub-features but the connection between a feature and its sub-features is mediated by the *group*. A group gathers together a set of features that are children features of some other feature and that are subject to a *local composition constraint* (see section 5.3). Thus, a group represents a cluster of features that are children of the same feature and that obey some constraint on their legal

combination. The same feature can have several groups attached to it. Both features and groups have cardinalities. The cardinality of a feature defines the number of instances of the feature that can appear in an application. The cardinality of a group defines the number of features chosen from within the group that can be instantiated in an application. Cardinalities can be expressed either as fixed values or as ranges of values. The distinction between group cardinality (the number of distinct features within the group that can be instantiated in an application) and feature cardinality (the number of times the feature can be instantiated in an application) avoids the multiplicity of representation and the consequent need for normalization found in some feature meta-models proposed by other authors.

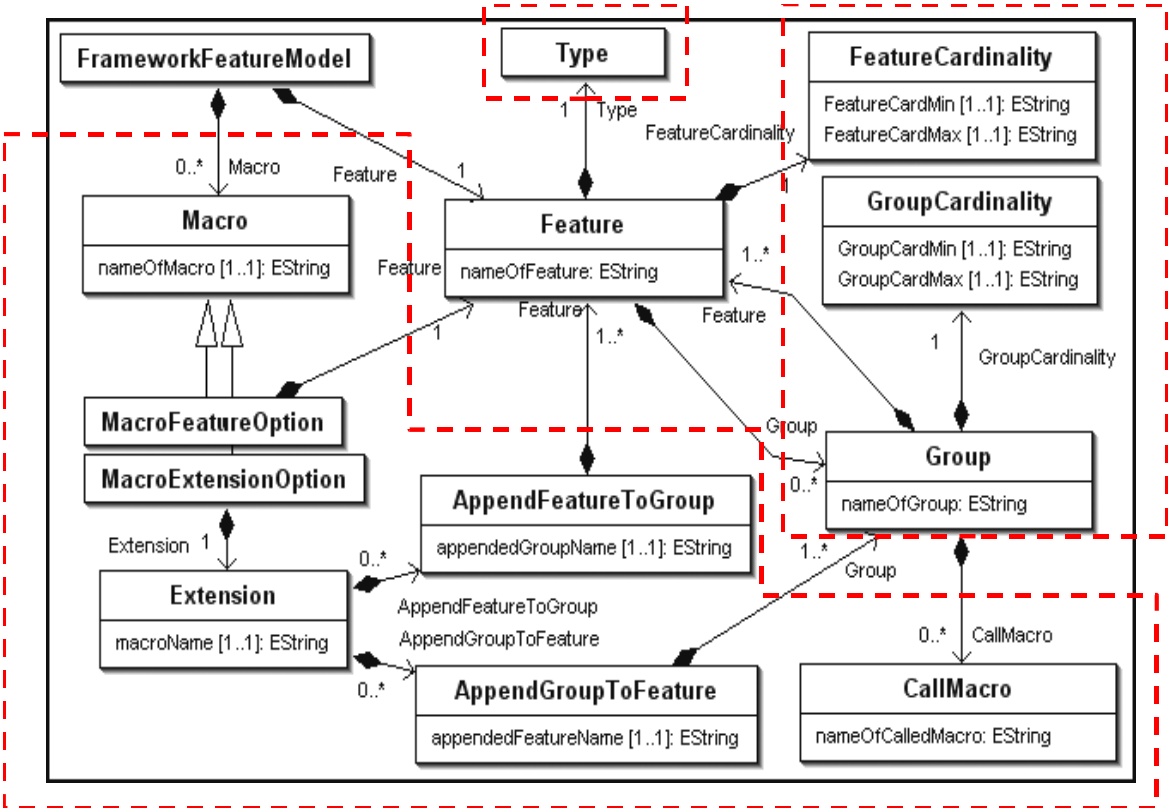

Fig. 12.1.1-1: ICSR Family Meta-Model

The items in the red contour on the left side at the bottom of the diagram of figure 12.1.1-1 cover the so-called *feature macro facility*. This is a mechanism to split a large feature diagram into smaller modules that can be used independently of each other. These modules are called *feature macros*. Feature macros resemble the macro facilities provided by some programming languages to encapsulate segments of code that can then be "rolled out" at several places in the same program. A feature macro represents a part of a feature diagram consisting of a node together with all its sub-nodes. The family meta-model allows a feature macro to be used wherever a feature can be used. A large feature diagram can thus be constructed as a set of independently developed modules. Note that the same feature macro can be used at different points in a feature diagram (see figure 12.1.1-2 for an example). Feature macros thus provide both modularity and reuse.

In order to enhance its reuse potential, the feature macro mechanism additionally provides an inheritance-like extension mechanism. Given a feature macro B (for "Base"), a second feature macro D (for "Derived") can be defined that extends B. Two mechanisms for extending feature macros are defined by the ICSR meta-model: (1) a derived feature macro D can add a new group to its base feature macro B, and (2) a derived feature macro D can add a new feature to one of the groups of its base feature macro B.

The macro mechanism allows a form of reuse where a feature sub-tree can be parameterized and instantiated for use in different parts of the same feature diagram with different requirements. This is more flexible than just allowing a feature sub-tree to be used in the same form at different places in a feature diagram. Note however that the analogy with inheritance is only partial because the ICSR meta-model does not offer the possibility of overriding existing sub-features in a base feature sub-tree: new sub-features can be added but existing ones cannot be deleted or modified.

Figure 12.1.1-2 shows an example of a feature model where the same feature macro is used at two different locations. The macro extension mechanism however is not used in the figure. The feature diagram in the figure is a fragment of the feature model briefly introduced in [Cec04.]

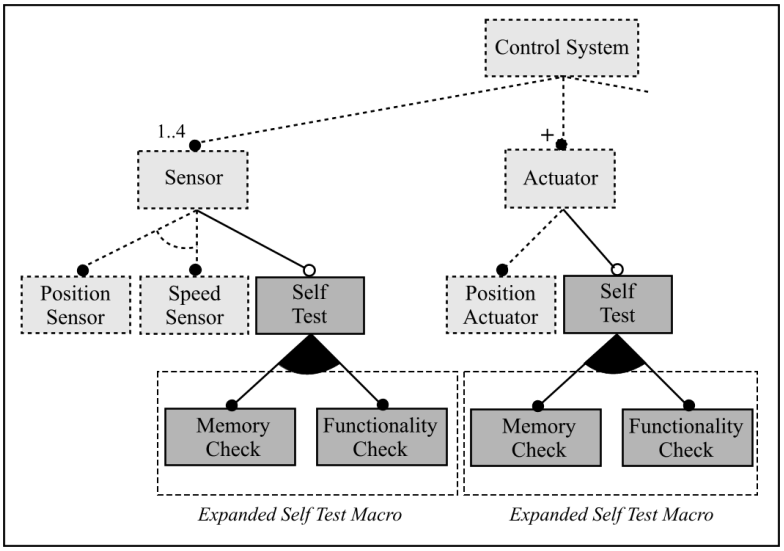

Fig. 12.1.1-2: Example of Feature Macro

Finally, the ICSR family meta-model attaches a type element to each feature. The structure of the type information really depends on the intended usage of the feature model. Indeed, one of the main benefits of the feature concept proposed in this document is precisely the flexibility it gives to the users to tune the structure of the type information to suit their special needs.

Possibly, the simplest kind of of type information is one that distinguishes between two types of features: *toggle features* or *valued features*. Toggle features are features that are either present or absent in an application. Valued features are features that, if they are present in an application, must have a value attached to them. Consider for instance the feature diagram of figure 4.1-1. The self-test feature is a toggle feature (a sensor is either capable of performing a self-test or it isn't). The memory size feature instead is a valued feature because its instantiation requires the application designer to specify a value for it (the actual internal memory size of the processor). The

type information discriminates between toggle and valued features and, in the case of valued features, it defines the type of the value.

Such a simple type structure is however inadequate to fully exercise the property set mechanism provided by the XFeature meta-meta-model. Hence, for demonstration purposes, a more complex feature type structure has been built into the ICSR model. This type structure is shown in figure 12.1.1-3.

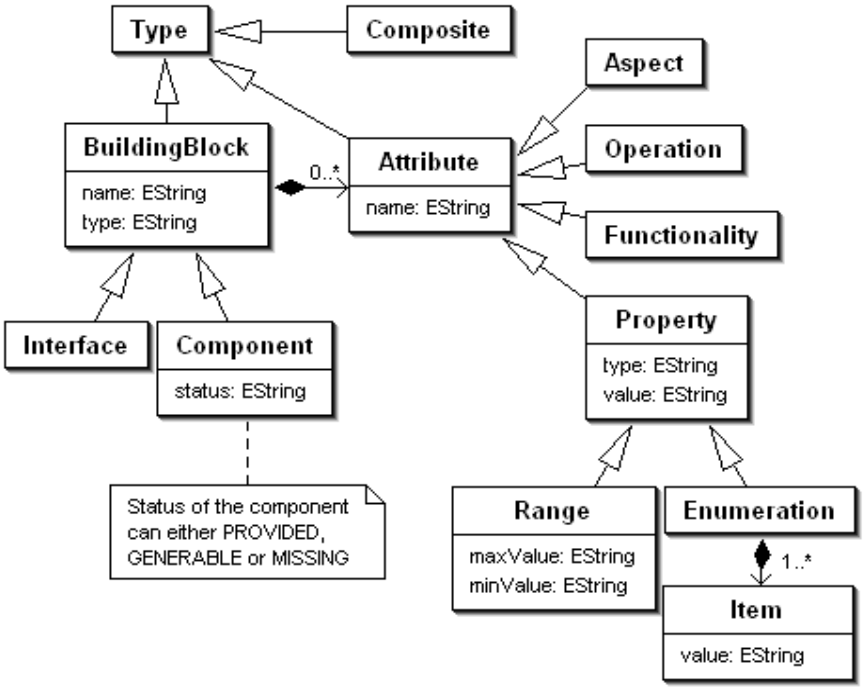

Fig. 12.1.1-3: Feature Type Structure for ICSR Model

The context within which the feature type structure of figure 12.1.1-3 was defined is that of software product family modelling. The feature model is used to describe the domain of a software product family. The feature model represents the set of features whose instantiation the software product family supports. The type information is used to map the feature to the software constructs in the product family. This information could be used to automate the family instantiation process.

The software assets offered by the software family are seen as a set of *building blocks* that can be used to construct applications in the family. Two kinds of building blocks are recognized: *components* and *abstract interfaces*. Building blocks have a *name* and are *typed*. Component building blocks additionally may have *attributes* that define their characteristics. Attributes too can be of different kinds. The simplest kind of attributes are *properties* (the term is used in the JavaBeans sense) but attributes can also represent *operations* that a building block may implement or they can represent *aspect,* i.e. cross cutting concerns that are shared by several types of components. The *functionality* attribute is provided to represent more abstract attributes that cannot be reduced to a property, an operation, or an aspect.

As was already mentioned, the objective of adding type information of the kind of figure 12.1.1-3 is to perform a mapping of the family features to specific software items provided by the family.

Obviously, some features may be too high-level to be thus mappable. For this reason, the type mechanism of figure 12.1.1-3 foresees the *composite* type.

Since the instantiation of an application will often require the development of application-specific components to complement the default components predefined by the family, the type mechanism of figure 12.1.1-3 assigns a *status* to each component. Components may be *provided* (the component is predefined by the family), *generable* (the component is not predefined by the family but the family offers a generator meta-component that can automatically generate it), or *missing* (the component must be developed anew).

The type information shown in figure 12.1.1-3 is entirely encoded using the property mechanism of the XFeature meta-meta-model and it demonstrates its versatility.

12.1.2 Application Meta-Model Generator

The ICSR application meta-model generator is built upon the premise that an application model is a feature model where all variability has been removed, namely as a feature model where all features are mandatory and where all features have cardinality of 1. In this sense, the feature meta-model generated by the application XSD generator is simpler than the meta-model of figure 12.1.2-1 because it does not include the group mechanism and the feature cardinality mechanisms. In another sense, however, it is more complex. The family model specifies the types of features that can appear in the applications and the local composition constraints to which they are subjected. The application XSD generator must express these constraints as an XML schema using the XSD language. Basically, this is done by mapping each feature group in the family meta-model to an XSD group and by constructing an XSD element for each legal combination of features in the group. This implies a combinatorial expansion and an exponential increase in the size of the application meta-models. It is, however, noteworthy that the computational time for applying the XML schema (i.e. the computation time for enforcing the application meta-model) only needs increase linearly with the number of features in the family model. The experience of the authors of this report with application meta-models derived from family models containing about a hundred features is that the computational time for enforcing the meta-model remains negligible and compliance with the meta-model can consequently be checked in real-time and continuously as the user selects new application features within a standard XML tool.

The type information is preserved and features at application level keep the type that was defined for them at family level.

12.1.3 Display Model

The display model defines how a feature model is to be rendered visually by the XFeature tool. The visual rendering is in turn defined in terms of *nodes* and their *properties*. Thus, a display model must accomplish two tasks: (1) it must define a mapping of the feature model items to nodes and properties, and (2) it must define how nodes and properties are to be visually rendered.

Task (1) will be considered first. Basically, the ICSR display model maps groups and features to nodes and it maps the values associated to features to the node properties.

In order to illustrate the mapping of groups and features to nodes, consider first figure 12.1.3-1. This represents a family of cars defined by: (1) a car has between 4 and 5 wheels, (2) there are two types of wheels (A and B), and (3) a car may have a sun roof which can be either manual or electric (but not both).

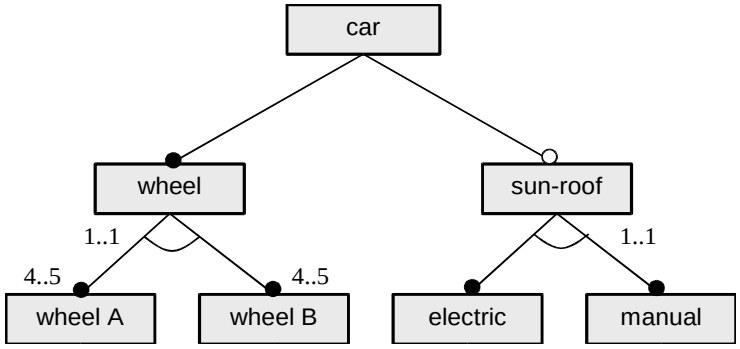

Fig. 12.1.3-1: Example of Family Model

From a conceptual point of view, the representation of this family model using the group/feature mechanism of the ICSR meta-model is as shown in figure 12.1.3-2. Each box in the figure represents a display node. Some display nodes represent features and other represent groups. The node cardinality represents either the feature or the group cardinality.

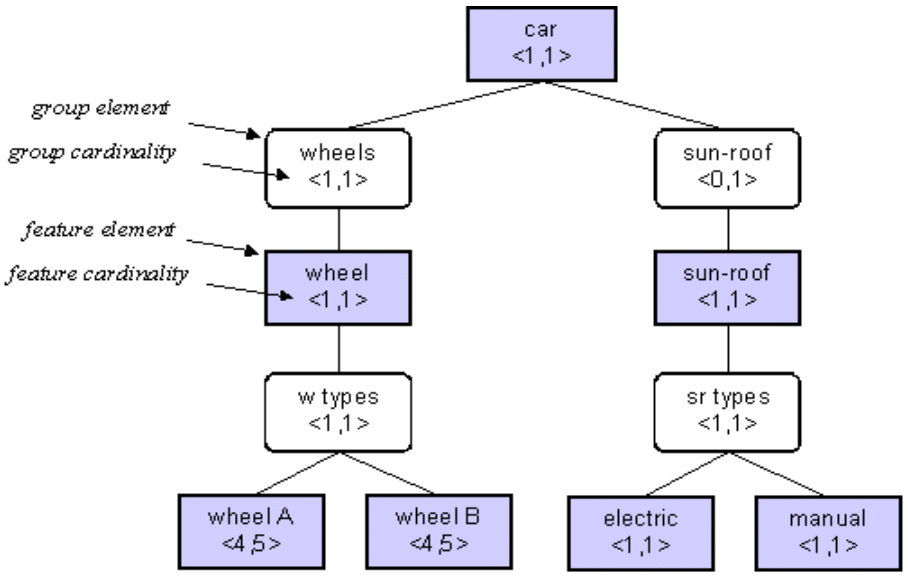

Fig. 12.1.3-2: Family Model of figure 12.1.3-1 according to ICSR Meta-Model

A second example may be helpful to further clarify the issue. Figure 12.1.3-3 shows the family model of figure 4.1-1 using the ICSR meta-model. The boxes in the figure again represents display nodes.

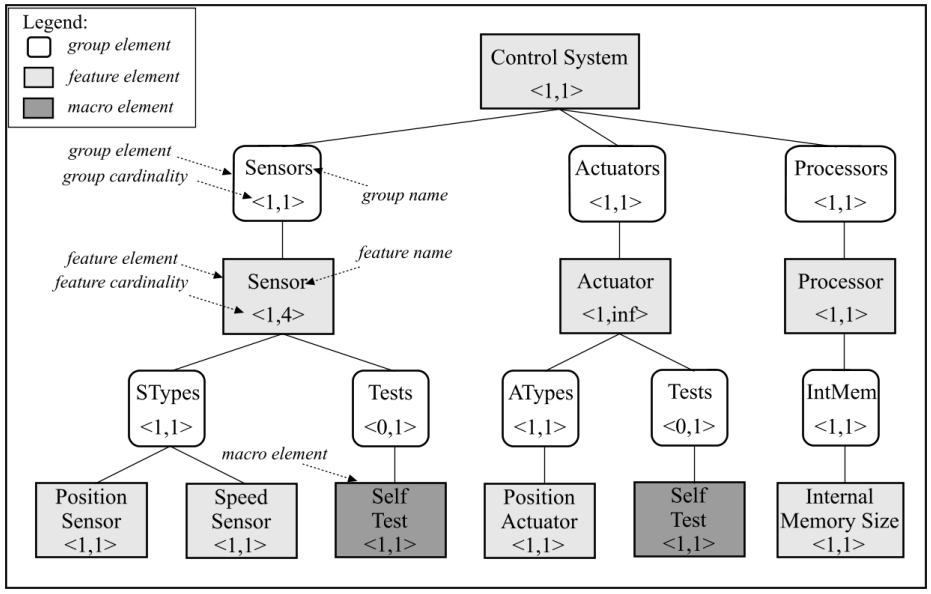

Fig. 12.1.3-3: Family Model of figure 4.1-1 according to ICSR Meta-Model

In addition to defining a mapping of the feature model items to nodes and properties, the display model must define how nodes and properties are to be visually rendered. In summary, this is done as follows. Feature nodes are rendered as rectangular boxes and group nodes are rendered as circular boxes. The connecting lines are defined such that the circles representing the group nodes are attached to the boxes representing the feature nodes. The cardinalities are shown as smaller boxes superimposed on the feature and group icons. Figure 12.1.3-4 shows how the family model of figures 4.1-1 and 12.1.3-3 is rendered using the ICSR display model.

Note finally that, from a display point of view, feature macros are treated as ordinary features with the only difference that they are rendered with a different colour.

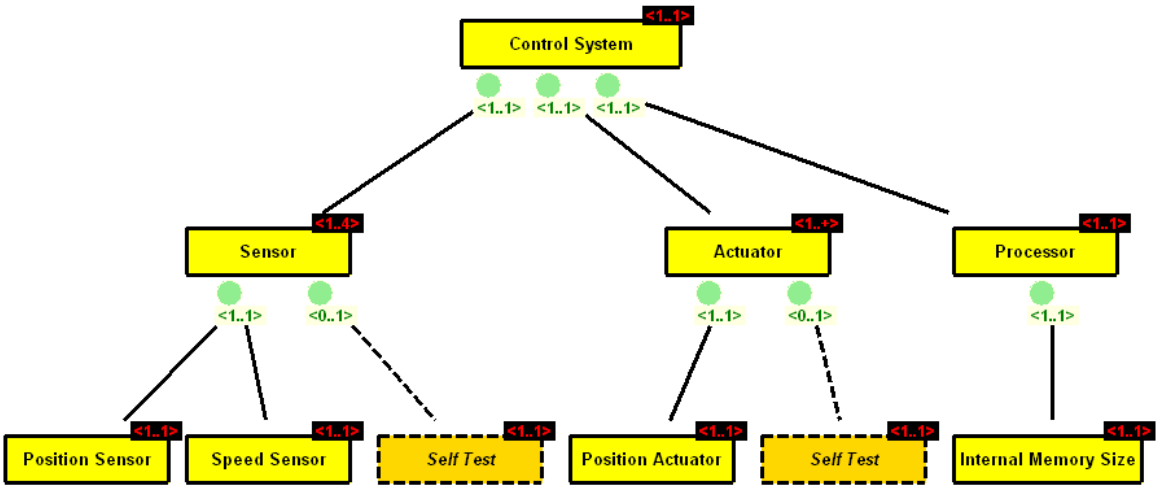

Fig. 12.1.3-4: Family Model of Figure 4.1-1 rendered according to ICSR Display Model

12.1.4 Application Display Model Generator

ICSR application models only consist of features. Figure 12.1.4-1 shows an example of a model of an application instantiated from the control system family of figure 4.1-1. The application model basically lists the feature that are present in the application.

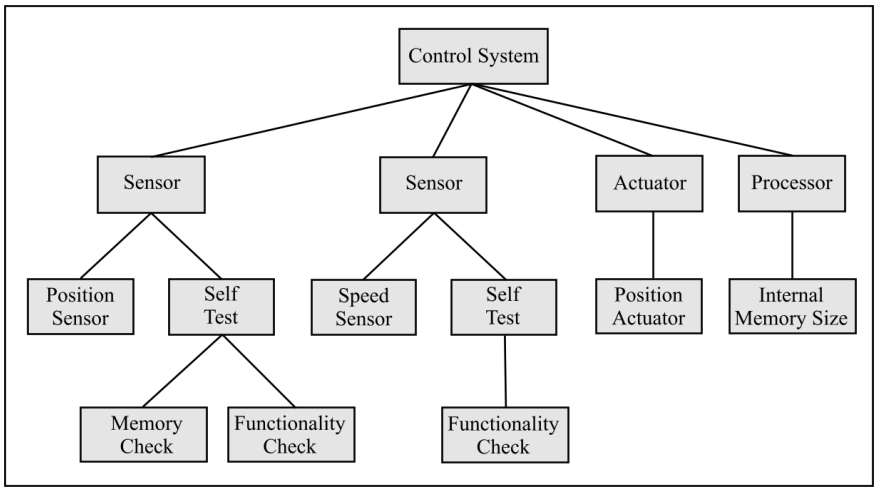

Fig. 12.1.4-1: Example of Application Model instantiated from Family of Figure 4.1-1

The application display model generator is very simple. It treats all features in the same manner and it uses the same rectangular box for all kinds of features. Figure 12.1.4-2 shows how the application model of figure 12.1.4-1 is rendered in the XFeature tool using the ICSR application display model.

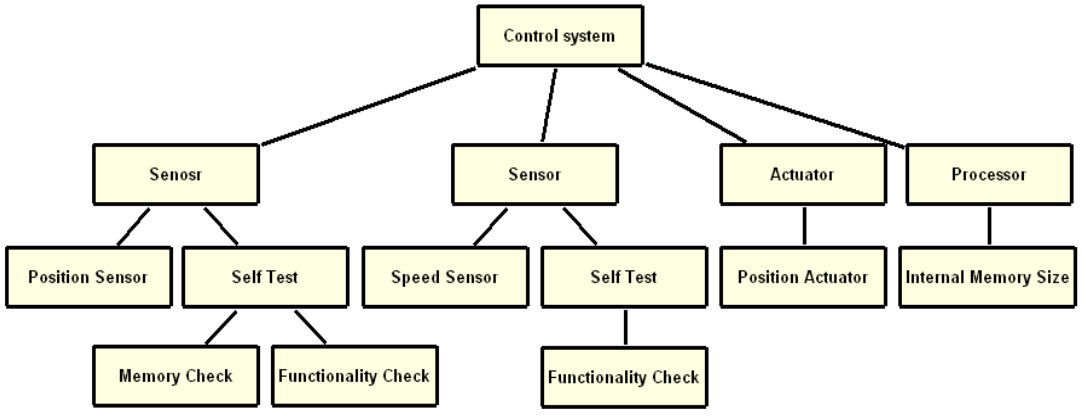

Fig. 12.1.4-2: XFeature Rendering of Application Model of Figure 12.1.4-1

12.2 The ICSR Simplified Default Configuration

The ICSR Simplified Default Configuration is obtained, as its name implies, as a simplification of the ICSR configuration. Its family meta-model is identical to the ICSR family meta-model with the only difference that it does not include the feature macro mechanism. The other configuration files are also derived from the respective ICSR configuration files after removal of the parts that cover feature macros.

Basically, the simplified ICSR family meta-model allows designers to define simple family models that are constituted of a single feature tree diagram. In this sense, it represents the simplest kind meaningful of family meta-model.