Metamodeling Editor as a Front End Tool for a CASE Shell

Matti Rossi*[1]
Mats Gustafsson[2]
Kari Smolander*[1]
Lars-Åke Johansson[2]
Kalle Lyytinen[1]

Abstract. Customizable Computer Aided Software Engineering (CASE) tools, often called CASE shells, are penetrating in the market. CASE shells provide a flexible environment to support a variety of information systems development methods. CASE shells are often cumbersome to use and in practice few people can model and implement methods in them. To overcome these problems we have developed a graphical metamodeling environment called MetaEdit and a method modeling interface to the CASE shell RAMATIC. Using this interface the methodology engineer can develop graphical models in RAMATIC's model definition language and then easily generate the resource files that control the operations of RAMATIC. MetaEdit is used as a graphical front end tool to develop, fast and in user-friendly manner, method models that can then be supported using RAMATIC. In this paper we shortly present the MetaEdit tool and then describe how the interface operates by illustrating how a new method for RAMATIC is defined using MetaEdit's interface tool. The defined method is called TEMPORA-ER.

---

* This research was in part funded by the Technology Development Center of Finland and Academy of Finland

[1] Department of Computer Science and Information Systems, University of Jyväskylä P.O. Box 35, SF-40351 Jyväskylä, Finland.

[2] Swedish Institute for Systems Development, Box 14225, SW-40020 Göteborg, Sweden

1. Introduction

Computer Aided Systems/Software Engineering (CASE) has been taken into extensive use in several companies. Most companies do not choose, however, a strategy to incorporate existing methods into tools, more likely the methods are determined by the chosen tool. This is the most obvious, but not in all situations, the most successful tool adaptation strategy. Therefore, one important research area is to examine how to develop CASE environments which can cover a variety of methods, and thus can satisfy the differing needs of various organizations. One solution is to use CASE shells - tools, by which one can tailor new functionality (ie. new method support) into a CASE environment. With a CASE shell, an organization can more easily build computer supported methods for a given task or project. A *method engineer* (an expert that defines the new methods) is usually needed to do this work, however.

One problem in this method adaptation process is that most CASE shells are quite cumbersome to use for method definition because they provide only low level and primitive mechanisms for this task. Therefore a limited number of people can actually model methods using CASE shells. To overcome this problem, one solution as proposed by Smolander et al. [10] is to use graphical tools for methodology modeling, i.e. graphical front-end tools for metamodeling, which we call metamodeling editors. These tools can be interfaced with different CASE shells and provide higher level mechanisms to accomplish the metamodeling task. The motivation to use a graphical model of method is to provide an abstraction layer that hides the peculiarities of a textual specification of the method.

We have developed a prototype metamodeling editor called MetaEdit [10]. In this study we discuss the general principles and architectures of metamodeling tools and demonstrate how we applied MetaEdit to specify methods into the CASE shell RAMATIC using its own specification language CML [1]. Our goal was to build an interface – a "bridge" – between MetaEdit and RAMATIC. This "bridge2 provides the sufficient functionality that allows a method engineer to graphically model a method in MetaEdit and then to create a text file which can be loaded into RAMATIC. The method engineer can concentrate on the critical tasks of method development such as method structure and content and let the tool handle the cumbersome production of a formal method specification that RAMATIC requires.

The paper is organized in six sections. The next section gives an overview of research that attempts to combine the CASE tools and provides motivation for our work. In the third section we shortly describe RAMATIC´s architecture. In the fourth section an overview of MetaEdit is presented with a more detailed description of the ReportGenerator that was used to implement the transformation system. The fifth section describes the functionality and structure of the bridge developed between

MetaEdit and RAMATIC. We also demonstrate how the modeling method TEMPORA-ER can be defined using this bridge. The final section suggests some future research issues and summarizes our results.

2. Connecting a Metamodeling Editor and a CASE Shell

2.1. Basic Preliminaries

Our interest in developing a metamodeling editor lies in our aim to model the methods used by the IS development groups to describe IS and it's environment. IS modelling is carried out by using a certain language, often referred to as a "description language". By a *method* we mean a set of steps and a set of rules that define how a representation of an IS is derived/transformed using a description language [9]. The "users" that define new description languages (*method engineers*) need corresponding languages (*meta languages*) to derive representations of the methods under development. These representations form a (IS) *metamodel* and the process of creating the meta model is called *metamodeling*.

The most obvious benefits of metamodeling are achieved[1] if we have a platform or a tool where the modeled method can be implemented. Such tools are called *CASE shells* and they offer mechanisms to specify a CASE tool for an arbitrary method or a chain of methods [4]. The concept of a CASE shell means that the shell can "learn" about methods which it did not "know" before. This learning will result in a tool that gives support for the use of the specified new method. To achieve this, one has to describe the method to the tool by defining the "method concepts" or design the object types of a specific description language (metalanguage). Meta concepts of a specific method may, for example, be objects, attributes, relationships, business functions, organizational units, information flows or whatever the methodology assumes to be useful in modeling the IS.

The meta languages vary from one CASE shell to another which makes their use difficult. Another problem is that often they use rigid and complicated textual languages which can be difficult to learn and apply. Therefore there is a need for more advanced method specification languages and associated support environments, which we call metamodeling tools or editors. A *metamodeling editor* is a special kind of CASE shell (meta tool), with which a method engineer can define methods for other (target) CASE shells graphically. In other words it can be "populated" by a set of meta languages used in different CASE shells.

---

[1] What are benefits if such an environment is not available are discussed in [3]. They are for example the concise description of methods, assesment of techniques and comparison of methods

The three main motivations in carrying out this study were: 1) there is a growing need for graphical (meta) modeling environments, 2) the best use of the graphical metamodels could be achieved if they can be transformed into the textual metalanguage of a given CASE shell by the metamodeling editor, and 3) the "programmability" of the metamodeling editor (meta CASE tool) gives us a chance to develop bridges into a number of CASE shells. The first item is obvious (see [10]) and also supported by the growing number of CASE shells in the market. The other two issues form the topic of the research reported in this paper.

2.2. The Motivation for Using Metamodeling Editors

The rationale behind using metamodeling editors is to provide a graphical environment for the methodology modeling. They are aimed to be general metamodeling environments which can be interfaced to different types of CASE shells with the appropriate "bridges".

One advantage of using graphical editors in metamodeling is that most of the meta models are easier to understand and maintain in a graphical form. Complex relationships between the concepts are easier to understand in pictorial form, than as lines of a textual definition. The maintenance, manipulation and modification of methods will also become easier when the meta models are kept in a graphical form. This allows for rapid modification of versions of meta models for different development situations and helps to improve their consistency and integrity. The possibility for versioning is essential in developing new methods, because method development tends to be cyclical and driven by method of trial and error [9,11]. This can be more easily achieved in computer environment where supported mechanisms are readily available (cf. version control systems). Finally, a benefit of graphical metamodeling is that it enhances the visibility of method development and gives the tool users a better understanding of the methods they use.

2.3. The Motivation for Using RAMATIC as a Target CASE Shell

The tool RAMATIC was chosen to act as the pilot CASE shell for interfacing because it has a large set of modeling constructs which form a "representative" example of those applied in other CASE shells. Hence, if we can "construct a bridge" for RAMATIC, we can probably implement it for other CASE shells as well.

In another study we have made a comparison between three CASE shells [7]. Results of the comparison of CASE shells show that RAMATIC has a powerful metamodeling language, but the method definition is rather difficult when compared to for example Excelerator´s Customizer [7]. The difficulties arise from CML´s rich set of concepts and their complex interdependencies. By using MetaEdit as a graphical interface we hope to achieve the best of both worlds: a powerful modeling language and the ease of using the

graphical modeling. The functionality of bridge is also needed as there is not much sense in modeling the methods in RAMATIC's graphical language without being able to automatically transform them for the RAMATIC's native schema language.

2.4. Modeling the Connection Between Metamodeling Editors and CASE Shells

In order to model methods in a metamodeling editor for a given CASE shell the metamodeling editor has to provide support for the native description language contained in the CASE shell. This requires that the definition of the CASE shell's language (called the meta-metamodel of the CASE shell) has to be defined in the description language of the metamodeling editor (it's metalanguage). To make the bridge operational, the method engineer also has to define an output specification of how the metamodeling editor's graphical model will be translated into the CASE shell's native textual language. This output (report) specification and the metamodel specification together form the "bridge" between the two tools. The benefit of a two sided connection between tools is that it allows for both the use of a graphical specification method in the metamodeling tool and the seamless transportation of the resulting specification into the CASE shell. The mappings between the tools are illustrated in figure 1.

On the right hand side of the figure we represent the three levels of languages that are needed in delivering the functionality of a CASE shell. The IS specification level is the end user level where methods are used to develop IS representations. The syntax and the "semantics" of these models are defined on the metamodel level.

These metamodels limit and guide the usage of the tool. The metamodels themselves are in turn based on a modeling language and its presentation form (syntax and semantics). This language level is called the meta-metamodel of the CASE shell. This level is important as it is instrumental in offering the flexibility of the CASE shell environment. Therefore the expressive power and usability are important goals in developing these meta-meta languages. One example of a language developed for this level is RAMATIC's CML language which will be discussed in section 3.

The same three levels are also present in the metamodeling editor and they form the left hand side of figure 1. Analogously, in a metamodeling editor we have the meta-metamodel, the metamodel and the model. These operate, however, on one abstraction lever higher than in the CASE shell. Thus, the model level in a metamodeling editor defines the structure and the functionality of the IS modeling language i.e. the metamodel of the CASE shell. The metamodel in a metamodeling editor is the model that is used for specifying the methods in CASE shell i.e. the meta-metamodel of the CASE shell.

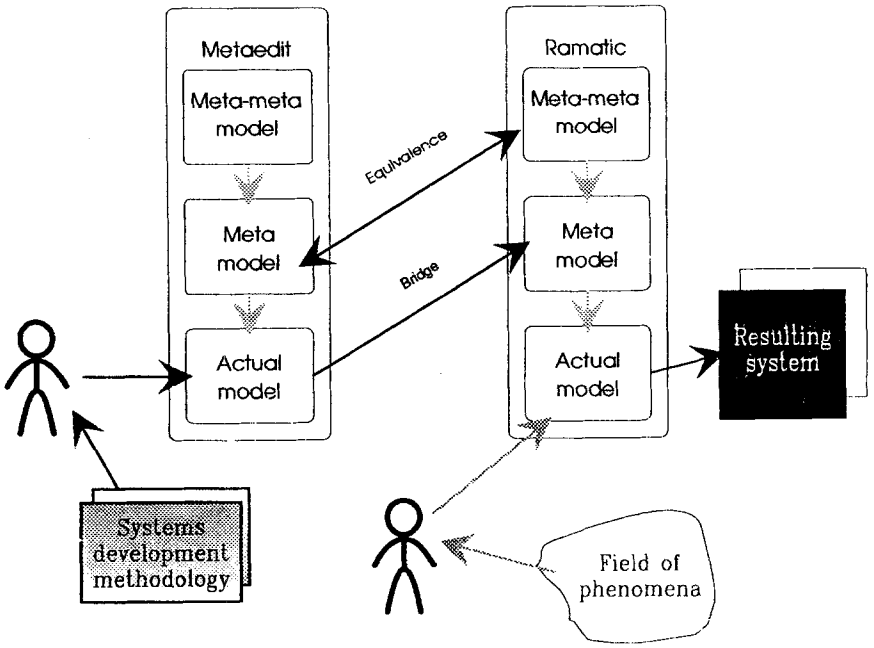

Fig 1. Mappings between MetaEdit and the target CASE shell

The third level defines the syntax and semantics of the metamodels used in the editor. The metamodels are used in specifying the method specifications for CASE shells. This highest level data model leverages the flexibility for the metamodeling editor. It allows users to define the CASE shell´s metamodels in its "own metalanguage". This is achieved by defining the meta-metamodel of the CASE shell using the meta-metamodel of the metamodeling editor.

The connections between the tools are depicted by the two arrows in the figure 1. The upper arrow pointing from a CASE shell to the metamodeling editor represents a mapping of a CASE shell´s meta-metalanguage into the metamodeling editor´s language. The upper arrow from the viewpoint of the metamodeling editor represents a mapping of the metamodeling editor´s language into the CASE shell´s language. It involves functions defining the transformation between representation forms. The lower arrow represents the functional part of the bridge and it shows how the models derived using the metamodeling editor are then transformed into a CASE shell´s metamodel. To demonstrate the viability of this approach, we will demonstrate how the metamodeling editor MetaEdit is used to translate graphical specifications into RAMATIC´s textual metamodels. This is discussed in the section 5. Before this we shall shortly describe the functions and architecture of both MetaEdit and the CASE shell RAMATIC. This is done in sections 3 and 4, respectively.

3. The Architecture of the Ramatic CASE Shell

3.1. An Overview of RAMATIC

RAMATIC is a CASE shell - a "meta-tool" developed by the Swedish Institute for Systems Development (SISU). The development of RAMATIC started in 1985 with the objective to create a flexible environment for prototyping CASE tools for a large number of different methods employed by the different organizations supporting SISU.

3.2. The Architecture of RAMATIC

The architecture and functions of RAMATIC are shown in figure 2. The core of RAMATIC is the design object database (*DODB*). *DODB* consists of two parts: the conceptual database (*CDB*) and the spatial database (*SDB*). *CDB* stores information about the developed methods for example their objects, the relationships between the objects and their attributes. *SDB* contains information of how and where on the screen the conceptual objects are graphically represented.

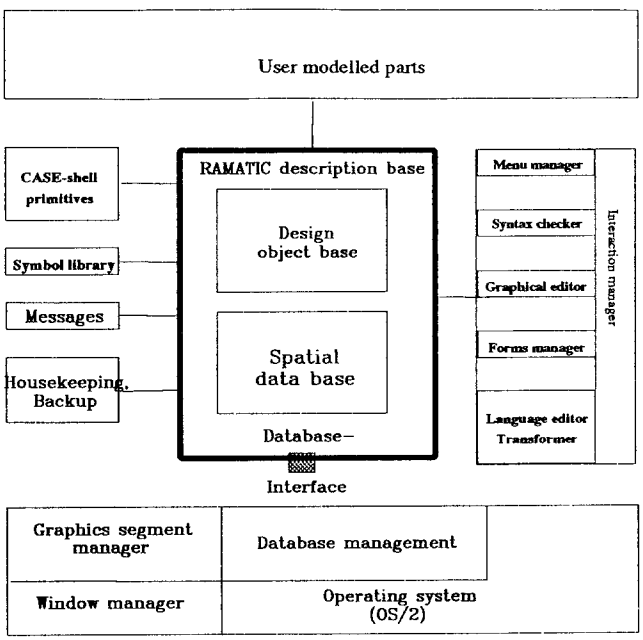

Fig 2. Architecture of RAMATIC

3.3. The Meta-metalevel of RAMATIC

RAMATIC uses a small set of meta-meta objects. These objects can have attributes which can be given values. The defined meta concepts and other definitions are stored in the "method knowledge base". As noticed above this base controls the uses of the tool in creating and manipulating design objects, according to the followed method.

In RAMATIC also graphical symbols can be defined on the metalevel by means of a symbol description language. This can be thought of as modifying the attributes of a "meta-meta symbol object". Design object types having graphical representation are then associated to one, or several symbol types.

3.4. Representation of Design Objects in RAMATIC

In RAMATIC certain design objects can exist independently of other objects, such as an entity type can exist as a "free" node in the ER-modeling. Other design objects, binary "relationships", can only exist if they are "connected" to two other object types. Such a relationship is defined as a "connection" design object. Accordingly, in a metamodel describing a business modeling technique we can recognize an information flow between two functions. This will be defined as a connection design object in RAMATIC. Attribute types for design object types can be defined as free nodes in RAMATIC. Alternatively they can be defined also as "subnodes" to the FREE object node. If we define these as FREE nodes, we can add more (meta)attributes to the attribute types.

The graphical representation of the objects is specified by a symbol definition. The symbols can be defined in terms of their form, size, color, line width and so on in RAMATIC. This is done through a separate symbol definition language. Notice, however, that we can define a meta concept that has no graphical representation.

After this short introduction we can examine the metalanguage of RAMATIC in more detail. As noticed above RAMATIC´s design objects are classified into:

- free meta concepts (FREE), and
- connecting meta concepts (CONN).

FREE- and CONN objects have attributes TYPE, ID, NAME and TEXT. The TYPE attribute indicates the type of the meta concept, the ID, NAME and TEXT attributes are used for names of concepts. TEXT is used exclusively for meta concepts having graphical representation.

CONN concepts connect pairs of FREE design objects by means of associations FROM and TO. CONN objects can, but need not, to have a graphical representation. The types of every valid pair of FREE objects must be defined for any CONN object type. The existence of any CONN object is dependent on the existence of those objects it connects. FREE and CONN objects can be defined also to possess attribute associations through an ASSOC statement.

For both FREE and CONN object types a SYMBOL TYPE can be attached. Moreover FREE- and CONN type objects can be grouped together by means of group (SET) objects.

Finally the metaconcepts of a specific modeling technique (these are usually handled in a separate session of the tool) are grouped together in one MODELTYPE. Meta concepts that appear in more than one model type are declared by INTER concepts. FREE and CONN objects of the same type may have IDENT associations. Furthermore, it is possible to define a set of integrity constraints for the design objects within one model type group. These constraints specify mostly the minimum and/or maximum cardinalities for the TO- and FROM associations. The constraints are formulated as FAULT expressions.

As an example of RAMATIC specification consider the following part of a definition of the meta concepts in an ER modeling technique developed by the ESPRIT project Tempora [12]. The Tempora ER model type constitutes an extended entity-relationship modeling technique with composite objects and time modeling.

```
MODELTYPE TEMPORA-ER
FREE ET   TPETRC   ID=NA NAME=MANDATORY NAME=UPPER
FREE ETT  TPETTRC  ID=MANDATORY NAME=MANDATORY NAME=UPPER
FREE DET  TPDETRC  ID=NA NAME=MANDATORY NAME=UPPER
FREE AVT  TPAVTRC  ID=NA NAME=MANDATORY NAME=UPPER
FREE RS   TPRSSQ   ID=NA NAME=NA
......
```

Examples of FREE design object types are here: Entity Type (ET), Timestamped Entity Type (ETT), Derived Entity Type (DET), Aggregate Value Type (AVT),Relationship (RS).

```
IDENT AVT
IDENT SVT
INTER ET
```

Aggregate value types and simple value types may have IDENT associations, which suggests a way of making multiple instances within one diagram possible. The INTER permits the connection of entities in one model type with for example data stores in another model type.

```
CONN BA  TPBAA  ET  RS   ID=OPTIONAL NAME=MANDATORY NAME=LOWER
CONN BA  TPBAA  ET  RST  ID=OPTIONAL NAME=MANDATORY NAME=LOWER
CONN BA  TPBAA  ET  DRS  ID=OPTIONAL NAME=MANDATORY NAME=LOWER
CONN BA  TPBAA  ET  DRST ID=OPTIONAL NAME=MANDATORY NAME=LOWER
......
```

The Binary Association (BA) CONN object type can connect an entity type to a relationship, to a timestamped relationship, to a derived relationship or to a timestamped derived relationship, etc. The line symbol (symbol type TPBAA) must, according to this example definition, be drawn from the entity type symbol to the relationship symbol i.e in the direction that the name attached to it suggests.

```
ASSOC ET DEFDATE
ASSOC ET CARDLTY
```

In addition to the TYPE, ID, NAME and TEXT attributes, an entity type may have additional attributes, such as definition date (DEFDATE) and cardinality (CARDLTY).

3.5. Specification of Methods Symbols, Menus and Forms in RAMATIC

The symbol types connected to the metaconcepts are defined in a separate part of the method specification base. This is accomplished through the symbol definition language. As a part of RAMATIC, a symbol library is available.

Any menu in RAMATIC is made up of several menu items, which can be texts or symbols. For each menu item, the following is defined:

- the text (TEXT) to appear in the menu, or if it is a symbol
- the symbol type (SYMBOL) and
- the shell function to be executed (MENUNR)

For the manipulation of non-graphical expressions RAMATIC provides forms in which one can modify and manipulate specific values of design objects. On the form definition level of RAMATIC one can define forms, how they should look like, how the field values are derived (from design objects), and how the tool should check the entered data values, etc.

4. The Architecture and Functions of MetaEdit

MetaEdit is a graphical metamodeling editor, a flexible methodology modeling environment. It can be interfaced with several CASE shells and thereby it can be populated with several metamodeling approaches. Moreover, it offers a graphical interface to carry out methodology modeling, and thereby it offers some advantages over the earlier environments. [10]

4.1. Functions

MetaEdit consists of three major functional components (see fig. 3):

1. **Main Window** offers the file and specification management functions. The selection of the modeling methodology and the maintenance of specifications in the methodology specification base (MSB) are done here. Other utilities of MetaEdit are also controlled from here.

2. **Draw Window** provides drawing functions to draw and edit specifications. It is generic and its behavior varies depending on the metamodel it uses.

3. **Output Generator** provides a programmable utility that helps to create reports, generate code or retrieve data from the methodology specification base. The tool

manager specifies the output specifications in the output specification base. The output specification base is a set of text files containing output generator code.

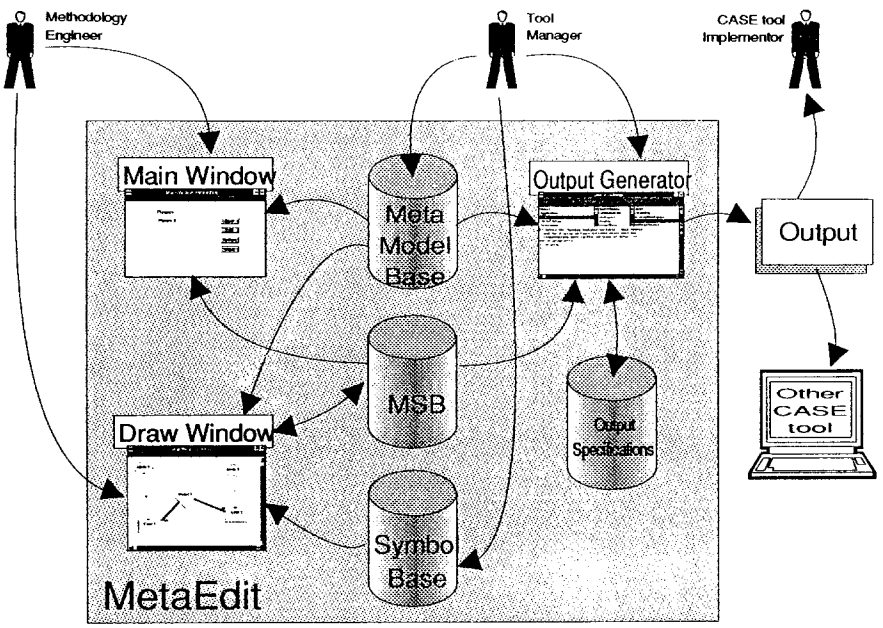

Fig 3. The Functional structure of MetaEdit

4.2. Datamodel

In MetaEdit the meta level datamodel is a fixed data structure based on the OPRR[2] data model [13]. The following defines the basic OPRR model :

Object is a "thing" which exists on its own. It is represented by its associated properties.

Property is a describing/qualifying characteristic associated with other object types (object, relationship, or role).

Role is a link between an object and a relationship. A role may have properties that clarify the way in which "things" participate in a certain part of a relationship. The role defines what "part" an object plays in a relationship.

Relationship is an association between two or more objects. It cannot exist without its associated objects. Relationships can also have properties.

The objects are always presented by graphical symbols in MetaEdit. The objects participate in relationships in certain roles. The roles are represented by symbols too. These symbols form the ends of relationship lines (for example arrow heads).

[2]OPRR stands for **O**bject, **P**roperty, **R**ole, **R**elationship model

Relationships are always represented by lines between two objects. Properties are represented by data fields. A field and it´s acceptable values are defined by a data type.

4.3. Model Definition

The meta-metalevel is used as a basis for all metamodel specification in MetaEdit. Depending on its instantiation, MetaEdit can be applied to different types of modeling approaches. Target level instances and their possible associations are fully determined by the definitions in the metamodel that can be changed on a user´s request at any time. Section 5 gives an example how we used the OPRR model to model the concepts of RAMATIC and thereby to instantiate the RAMATIC´s metamodeling approach. The detailed syntax of the metalanguage is described in [11].

4.4. Report Generation

To produce various types of output from MetaEdit´s models we have developed a general purpose report generator. It consists of two parts: ReportDesigner (a tool for building report specifications) and ReportGenerator, the actual machine to produce reports based on the stored definitions. The ReportDesigner is primarily intended for defining bridges from MetaEdit, but it can also be used to create integrity checking mechanisms for method specifications and to produce human readable documents.

The concept of a programmable transformation generator is similar to ideas in some other CASE shells. For example the Metaview environment has a transformation system for modelling transformations between model types [2] and Chen [4 pp. 130-136] has proposed a very similar environment to the ReportDesigner for a transformation language definition.

The ReportDesigner features an object-oriented query language, based on the syntax of Actor[3] language. The language has predefined functions for selecting objects from MetaEdit´s methodology specification base and for retrieving properties, roles and relationships of selected objects. The queries are constructed from these predefined functions by combining them into query methods. Each report specification contains one or more query methods that are joined together within a main function.

The ReportDesigner offers a Smalltalk[tm] style query browser. The user interface of the ReportDesigner is shown in Figure 4. It consists of four window portions within a main window. A report class window offers a list of defined reports. It is located in the upper left corner of the main window. When one of the report names is selected, the queries defined for that report are shown in the upper right-hand window, the query window. Again, when one of the queries is selected from the query window, its code is shown in

---

[3] Actor is a trademark of the Whitewater group

the edit window below. Between the report and query windows there is a model window where the constructs of the current metamodel are presented.

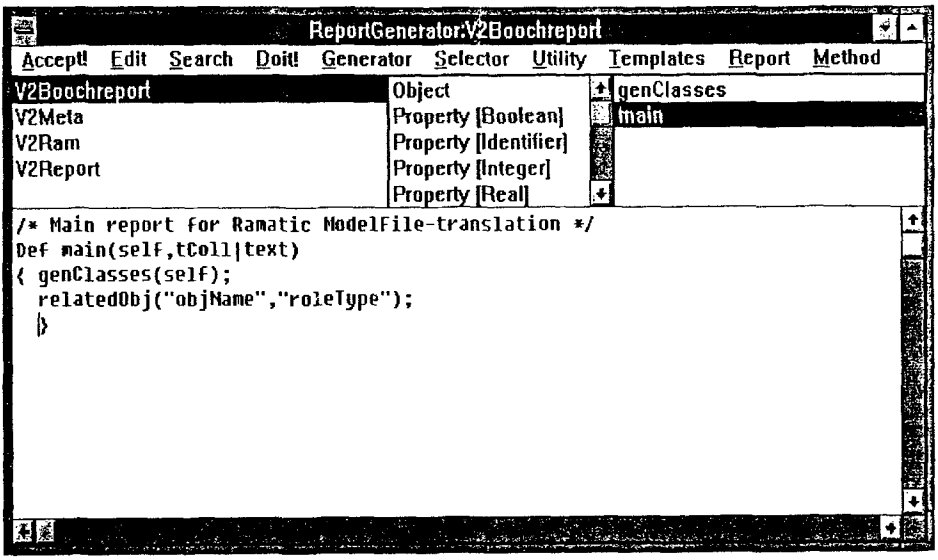

Fig. 4. The Interface of ReportDesigner

When the report specification is defined and compiled, it can be run in the ReportGenerator against the methodology specification base. The report generator can direct the generated reports to requested output devices while performing the requested query. The user can modify the behavior of the ReportGenerator by changing the report generator primitives in the ReportDesigner.

5. The Bridge Between MetaEdit and RAMATIC

This section presents the implementation of a graphical metamodeling editor for RAMATIC using MetaEdit. As the first step we defined RAMATIC´s meta-metamodel within MetaEdit using it´s OPRR metamodel. The mappings of RAMATIC´s design objects to MetaEdit´s OPRR constructs is presented in the next subsection. As the second step we define the report functions that represent the transformations from OPRR to RAMATIC´s CML.

5.1. Implementing RAMATIC´s Model Definition Language Within MetaEdit

We defined RAMATIC´s meta-metamodel in terms of MetaEdit´s OPRR meta-metamodel (the upper arrow pointing left in Figure 1). This step was accomplished by examining the design objects of RAMATIC and their properties and expressing them

with OPRR constructs. All the roles and relationships embedded in RAMATIC's CML had also to be defined. For all object types we defined the representation that specifies the graphical appearance of the CML object.

Based on the description of RAMATIC's modeling language (see section 3) we can recognise the following object types in OPRR: FREE, CONN, ASSOC and SET (Note that the relationship CONN is also represented as an object). These objects are connected to one another by the relationship types Connected to, Connected from, Has assoc, is Member and is Owner. For example, a FREE object type can be in the roles of Free part in the Connected to and Connected from relationships. This defines one end of the Connected to/from relationship in which a CONN object type is always in the role of Conn part. Accordingly the Assoc object-types can be connected to CONN and FREE object types in a has assoc relationship type.

All object types have the identifying property Type name. FREE and CONN objects have also the properties Symbol identifier, Set auto, Ident, Inter model name and ID and Name options. The fault options of RAMATIC are properties of the Connected to/from relationship types.

To be able to present the method graphically we also defined the symbols for object types and line types for relationships (see fig. 5). The properties are handled in a property dialog which is generated automatically to fit with the properties of an object-, role- or relationship type (see fig. 6).

Note that this is only a partial definition of the whole metamodel. A more thorough definition can be found in [8] and a part of it is in Appendix 1. The resulting meta model was tested and verified and used as a basis to specify the graphical interface with which all modeling information associated with RAMATIC's modeling language could be fed during the specification session. This formed the basis for the subsequent step to implement the transformation component of the bridge. In [10] we provide a more detailed description of the model definition in MetaEdit.

5.2. Mapping the MetaEdit's Design Objects to RAMATIC Model Definition Language

After specifying the modeling language and its representation forms in MetaEdit's OPRR constructs, the transformation problem could be expressed in MetaEdit's query language (The upper arrow in Figure 1 pointing to RAMATIC). The transformation task (method in object oriented vocabulary) could now be written as a series of query language commands by which the fixed expressions (reserved words used in RAMATIC's method specification) could be attached to the object instances derived

from a query to the specification base MSB. Hence this step produces the actual output for RAMATIC (bridge arrow in Figure 1).

In implementing this problem we had to define a translation method for each object type defined in the metamodel of MetaEdit. The principle of a translation method is as follows: first select the object instances for each object type and then specify how their properties (and relationships) are translated into the target language sentences. For example a method for transforming all FREE objects in MetaEdit into the corresponding FREE -lines in RAMATIC's specification does roughly the following:

```
For all FREE-objs
  do
    getproperties(Type name, Symbol identifier, Set auto, ID options,
Name options)
    getrelatedobject(Assoc)
    print("FREE")
    print(Type name, Symbol identifier, Set auto)
    print(Assoc name)
    print(ID options, Name options)
  enddo
```

The method first retrieves all FREE objects (first line) from the specification base, and then a set of named properties (third line) and relationships (fourth line) for each FREE object are retrieved. Next a fixed expression (fifth line) is printed to the output stream and the retrieved properties (sixth and eight line), and finally names of the related objects (seventh line) are added in a specific order.

The `Ident`, `Inter` and `Fault` command lines are derived from properties of the FREE- and CONN object types and therefore the methods for producing them are of a slightiy different form:

```
For all FREE-objs or CONN-objs with property(Ident)=TRUE
  do
    getproperty(Type name)
    print("IDENT")
    print(Type name)
  enddo
```

The difference is that now only a subset of FREE- and CONN objects with a certain value of a certain property are retrieved.

As most of the transformations are very straightforward the method engineer needs only to define the appropriate properties of the objects types and add some "syntactic sugar" to build a model definition clause in RAMATIC.

These examples shed some light on the main characterictics and the current status of the metamodeling editor and how it's query definition facilities can be used to build

effective bridges between MetaEdit and different CASE shells. The query formulation for producing FREE clauses in the CML language is shown in appendix 2. It points out that the current version of the query language uses rather complex syntax (if you do not like Smalltalk) and suits mostly for an experienced user. We believe, however, that the learning threshold is not a serious problem as long as MetaEdit is used mainly as a metamodeling editor, because on this level there are no "non-professional" users. The transformation task is for the first time quite tedious, but it has to be done only once and after that the query methods need only to be altered when the metamodel is restructured. Also the available query methods can be reused in creating new report specifications.

5.3. Using the Interface to Define the Tempora-ER Model

To demonstrate the usefulness of the metamodeling editor we used the editor to build a graphical model of the Tempora-ER model in RAMATIC's modeling language. Note that now the model of the Tempora-ER method is build graphically in RAMATIC's modeling language using the functionality of MetaEdit.

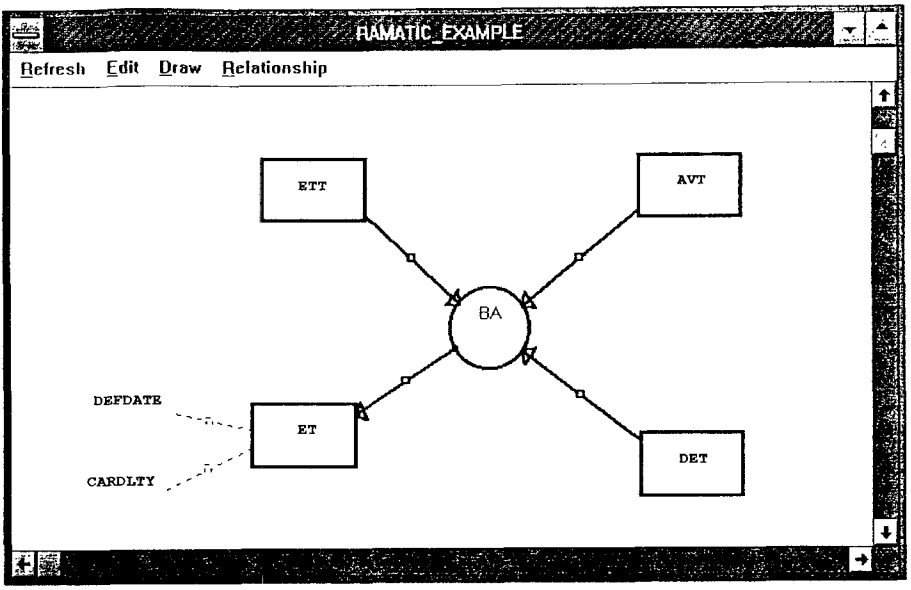

Fig 5. Tempora-ER in RAMATIC's graphical modeling language using the metamodeling editor

The DrawWindow of MetaEdit with the Tempora-ER model is presented in figure 5. The FREE design objects of Tempora-ER are ETT, AVT, ET and DET (See subsection 3.3 for details) and are represented by rectangles in Figure 5. There is one CONN object (the circle in the picture), BA, which has relationships with all the FREE objects. The ET object has two associations CARDLTY and DEFDATE (labels in Figure 5).

Figure 6 presents a property dialog for the ET -free object which is generated by using the OPRR specification of the FREE object. It represents properties of object types using different data fields. For example a FREE type has a text field, whereas a boolean type Ident property is represented by a check box, and the ID options by a list with predefined values (see appendix 1 for property definitions).

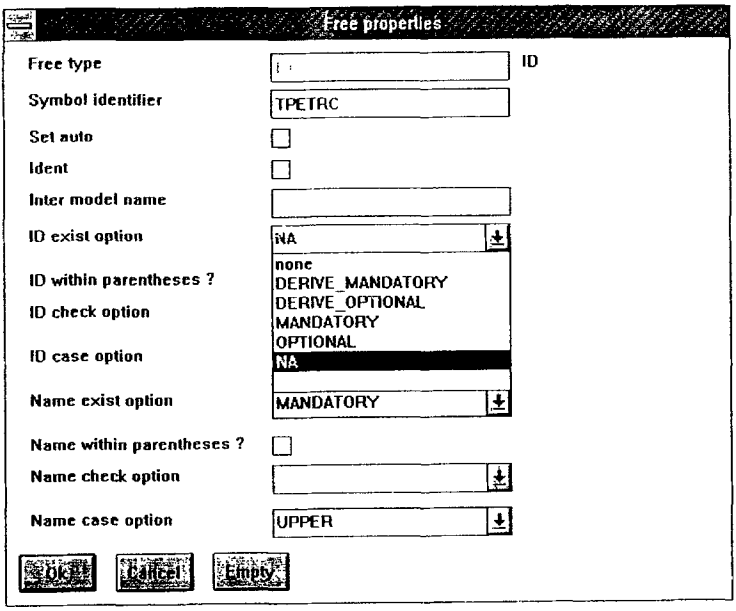

Fig 6. Property dialog for ET -object

Notice that this interface is generated automatically after we have specified the RAMATIC's modeling language in the OPRR notation and loaded it as metamodeling method into MetaEdit.

When the method engineer decides to produce a prototype of a report, in this example the TEMPORA-ER definition, s/he can run the report generator with the report specification for generating RAMATIC's modeling language constructs. For the user the ReportGenerator appears as a dialog that presents the user with a list of available reports and output devices. After selecting the RAMATIC report the user gets a method definition like that shown in Figure 7. This output is produced from the TEMPORA-ER model. The figure shows that all properties of the ER method have been successfully translated into the report output.

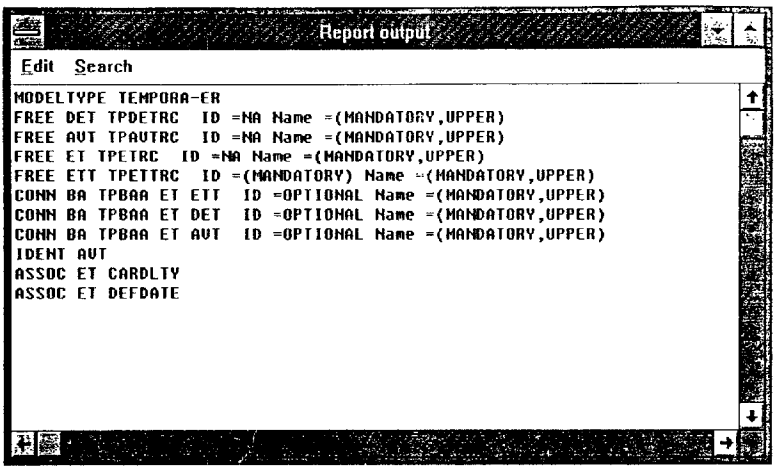

Fig 7. The generated report output of Tempora-ER

The example shows some of the strengths of the approach. A method engineer familiar with the basic concepts of RAMATIC's modeling approach can specify methods graphically, and then generate RAMATIC's textual method specifications. This relieves him/her from worrying about the syntactic details of RAMATIC's CML model definition language. The use of a metamodeling editor could lead into a situation where the method engineer builds the methods with method users, and technical RAMATIC specialists build the interfaces (symbols and forms) for them. Thereafter new versions of the method could be developed by changing parts of the method specification in MetaEdit, and producing then the output for RAMATIC.

6. CONCLUSIONS

In this paper we have presented how an interface between a graphical meta modeling editor and a CASE shell can be developed. Currently, the bridge is a research prototype and cannot produce all the files needed to define a method for the RAMATIC CASE shell. The bridge is, however, capable of producing complete conceptual base model definitions of RAMATIC. These can be used as a basis for defining other necessary parts of the CASE environment. This part of the method definition forms also the most crucial part in the specification of the method. The menu and symbol definition parts of RAMATIC are currently being redefined by SISU, and for that reason they were not included into the prototype. When they have been frozen we intend to cover them in a future version of the "bridge".

A full method development for RAMATIC using the metamodeling editor needs also the ability to model the interdependencies between models. These interdependencies can be handled by the transformation system or using links between models.

The most urgent research task in the future is to define a full-scale metamodeling editor that covers the remaining parts of the RAMATIC method definition. The benefit of such an extension will be that the time to build a CASE tool to support a particular method using RAMATIC will be a matter of days, as opposed to the approximation that it takes a few weeks to develop a tool by hand [1]. Another benefit from this is that the method developer needs only to know the concepts of RAMATIC´s modeling language and s/he does not have to worry about the "syntactic sugar" and the specific technical aspects of the language. Thus, a less experienced user can define methods for his/her own purposes and the learning curve will be steeper. In an optimal situation, one could make a model of a method with MetaEdit, produce some reports, load the new method definition into RAMATIC and try it out on the fly.

This graphical method modeling approach could be as well used for other CASE shell environments and in fact we have developed a similar bridge to develop graphically methods for MetaEdit itself. We claim that this approach would be desirable for commercial CASE shells as well.

MetaEdit´s current report generation facility is still a prototype environment. In the future we hope to develop a less awkward language for the query formulation. One promising possibility is to use the developed graphical model (such as RAMATIC´s graphical modeling language) in formulating queries. Despite its current limitations, the "bridge" demonstrates that graphical methodology specifications can be developed and automatically translated for use in a CASE shell. Hence, the modifiability of the metamodeling environment and its report generation facility gives method developers the possibility to deliver methods for specific CASE shell environments fast and effectively.

Acknowledgments

We would like to thank the referees and co-workers whose valuable comments greatly helped to improve the paper.

References

1. Bergsten, P., Bubenko, J., Dahl, R., Gustafsson, M., Johansson, L.-Å., RAMATIC - a CASE shell for implementation of specific CASE tools. TEMPORA T6.1, SISU, Stockholm, 1989. Draft of TEMPORA- report section 4.4.

2. Boloix, G., Sorenson, P., Tremblay, J., On transformations using a metasystem approach to software development, Dept of Computing Science, The University of Alberta, November 1991.

3. Brinkkemper, S., Formalization of Information Systems Modelling. PhD thesis, Thesis Publishers, University of Nijmegen, Nijmegen, Holland, 1990.

4. Bubenko, J., Selecting a strategy for Computer-Aided Software Engineering (CASE). SYSLAB-report n:o 59, University of Stockholm, Stockholm, Sweden, 1988.

5. Chen, M., The integration of Organization and Information Systems Modeling: A Metasystem Approach to the Generation of Group Decision Support Systems and Computer-Aided Software Engineering, University of Arizona, dissertation, 1988.

6. Dahl, R., RAMATIC description language, SISU, Sweden, 1990.

7. Marttiin, P., Rossi, M., Tahvanainen, V-P, Lyytinen, K., A Comparative Review of CASE shells, in Proceedings of the First Software Engineering Research Forum, Tampa, Florida, November 7 - 9, 1991, (ed. R. Rodriguez), University of West Florida, 1991.

8. MetaEdit user´s guide, RAMATIC model manual, Rossi, M. Smolander, K. Marttiin, P., Jyväskylä, 1991

9. Lyytinen, K., Smolander, K., Tahvanainen, V.-P., Modeling CASE environments in systems development. in Proceedings of CASE89 The First Nordic Conference on Advanced Systems Engineering, Stockholm, 1989.

10. Smolander, K., Lyytinen, K., Tahvanainen, V.-P., Marttiin P., MetaEdit - A flexible graphical environment for methodology modelling, in Advanced Information Systems Engineering, (eds. R. Andersen, J. Bubenko, A. Sølvberg), Springer-Verlag, 1991, pp. 168-193.

11. Smolander, K. OPRR - A model for modeling systems development methods, In Proceedings of the Second Workshop on The Next Generation of CASE Tools, Trondheim, Norway, May 11 - 12, 1990, (eds. V-P. Tahvanainen and K. Lyytinen), University of Jyväskylä, Jyväskylä, 1991

12. TEMPORA, Concepts Manual, Tempora ESPRIT project, E2469, September, 1990.

13. Welke, R., Metabase - A Platform for the next generation of meta systems products. In Proceedings of CASE Studies 1988, Meta Systems, Ann Arbor, May 23-27, 1988, Meta Systems, Meta Ref. #C8824, 1988.

Appendix 1. FREE-object Definition in OPRR Metalanguage

```
shape "Rectangle"
     {shape (0@40, 0@160, 200@160, 200@40, 0@40);
     line type "Solid";
     line width 3;
     connection points (100@160,0@160,0@100,0@40,100@40,200@40);}

symbol "FreeNode"
     {shapes ("Rectangle");
     scale 0.4;
     labels { "Free type" at (10 60 190 140) centered;}}

property type "Free type"
    {datatype String;
    values unique;}
property type "ID options"
    { datatype list("NA","OPTIONAL","MANDATORY","none");
    number of values 1;}
property type "Name options"
    { datatype list("DUPLICATE","UNIQUE","IDENTIFY","none");
    number of values 1;}
property type "Symbol identifier"
    {datatype String;
    number of values 1;}
property type "Set auto"
    {datatype Boolean;
    number of values 1;}

object type "Free"
{ symbol "FreeNode";
     duplicates not allowed;
     properties ("Free type", "Symbol identifier", "Set auto",
"Ident", "Inter model name","ID options","Name options" );}
```

Appendix 2. Code Example of FREE-Definition in MetaEdit´s Query Language

```
/* This is a report method for RAMATIC FREE-object generation */
Def genFrees(self|freeColl,rol,txt)
{ freeColl:=selObj(mme,"meta","Free",nil,nil,nil);
  do(freeColl,
  {using(elem)
    add(reportFile,"FREE "+getPropVal(elem,"Free type"));
    add(reportFile,getPropVal(elem,"Symbol identifier"));
    if getPropVal(elem,"Set auto")
       then add(reportFile,"SETAUTO ");
    endif;
    rol:=findRelRoles(elem,"add type","has Assoc");
    do(rol,
    {using(e)
      if getPropVal(theRelationship(e), "Add on create ?")
        then add(reportFile,"ADD"
getPropVal(relatedObj(e),"Assoc name"));
      endif;
    });
    endif;
    if getPropVal(elem,"ID qualifiers")
    then add(reportFile," ID=(",getPropVal(self,elem,"ID")),")");
    endif;
    if getPropVal(elem,"Name qualifiers")
       then add(reportFile,"
NAME=("+getPropVal(self,elem,"Name",',')+,")");
    endif;
    separator(self);
  });
```