MetaEdit–

A Flexible Graphical Environment for Methodology Modelling[1]

Kari Smolander
Kalle Lyytinen
Veli-Pekka Tahvanainen
Pentti Marttiin

Project Syti
Department of Computer Science
University of Jyväskylä
PL 35
SF-40351 JYVÄSKYLÄ
Finland

ABSTRACT

Existing CASE tools are often rigid and do not support the users' native methodologies. To alleviate this, more flexible and customisable tools called CASE shells are emerging. However, the customisation of those tools is still cumbersome and error-prone, and demands several configuration files that follow a rigid syntax of some metamodelling language(s). In order to make the customisation easier, we propose a graphical metamodelling editor, MetaEdit, with which the conceptual structures of the user methodology can be modelled easily using an easy-to-grasp graphical notation. With MetaEdit, methodology models can be constructed with less effort and the configuration files for the CASE shell can be created (semi)automatically. The tool is flexible i.e. its symbols and metamodel are user-definable. In consequence it can be used as a simple CASE shell. MetaEdit is based on the Object-Property-Role-Relationship (OPRR) data model. The paper presents the principles on which the editor is built, describes its operation, and discusses its relations to other research on metamodelling.

Keywords: Methodology, Computer Aided Software Engineering, Metamodelling, Methodology Engineering, CASE-shells.

---

[1]This research was in part funded by the Technology Development Center of Finland.

1. INTRODUCTION

Computer Aided Systems (Software) Engineering (CASE) has experienced a renaissance during the last five years (Chikofsky 1988, Penedo and Riddle 1988, Lockemann and Mayr 1986). New software products have been introduced (for surveys see e.g. Everest and Alanis 1989) and several researchers have debated on the concept of CASE (Chen *et al.* 1989; Lyytinen *et al.* 1989; Bubenko 1988). Also the impact of CASE on productivity, work-place (LeQuesne 1988; Orlikowski 1988, 1989) and management strategies (Siltanen 1990) has received growing attention.

However, despite the close connection of CASE tools to systems development methods and methodologies, much less has been examined how methods and CASE-tools can and should be matched with one another in systems work (for earlier work see Teichroew *et al.* 1980; Welke 1983, 1988; Kumar and Welke 1988; Venable and Truex 1988). This research issue opens up several challenging research problems such as:

(1) How should methods and tools interact with each other? Should the tools dictate the methods, or should the methods dictate the tools?

(2) If the methods should dictate the tools, how can one specify the tool environment so that methods of different species can be supported?

(3) If methods of varying sorts need to be supported in a tool environment how should the methods be modelled and specified for a given tool environment? What (meta)methods and (meta)tools are needed in this process?.

(4) What functionality and interfaces should such a (meta)tool environment offer?

Each of these questions has been discussed at some length in the literature.

Re 1) Bubenko (1988) suggests several generic approaches for matching tools and methods (see also Lyytinen 1988), and Smolander *et al.* (1990) report how successfully different approaches have been exploited in practice.

Re 2) The first generic tool environment that could support configured methods was SEM (System Encyclopedia Manager) by Yamamoto and Teichroew (Teichroew *et al.* 1980, see also ISDOS 1981). Since then similar approaches have been suggested by Sorenson *et al.* (1988) in their Metaview and Nunamaker and Konsynski in their Metaplex (Chen and Nunamaker 1989). All these approaches use a higher level (meta)language to define the abstract syntax of the specification language associated with a method. These tool environments focus on the storage and retrieval of linearised system descriptions i.e the

tool environments are heavily database oriented. RAMATIC (Bergsten *et al.* 1989) offers another road, where the tool environment is built around graphical notations associated with methods. Here the tool environment is constructed around a set of generic routines which can be associated with various symbols to be drawn i.e. the environment can be called interface oriented. A third, and a modest approach, has been to build extension kits for available methods supported by the CASE tool. This is the approach followed in customisers which are offered by several CASE distributors such as e.g. by Index Technology (1987).

Re 3) Usually the method is described for a tool environment by some conventional data model such as the E-R model (Teichroew *et al.* 1980). Due to the limited power of the E-R model, several extensions such as *is-part-of* and *is-kind-of* hierarchies (Chen and Nunamaker 1989), roles (Welke 1988) and set-functions (Bergsten *et al.* 1989), have been suggested to increase its semantic power. Another approach has been to develop mathematical formalisms such as graph theoretical foundations for more formal descriptions that can then be implemented (Harel 1988). However, little is known of the relative merits and disadvantages of various formalisms.

Re 4) Currently, the interfaces of metatools are cumbersome and based on a linear syntax. Oftentimes the specification involves an error prone and laborious manual compilation of the method "schema" which can include a plethora of implementation detail. Therefore, method development is a complex and difficult undertaking which requires high technical skills and good knowledge of the method being modelled and the tool environment — currently a scarce resource.

In this paper we present goals, motivation and basic principles of a metatool environment that tries to address some of the unresolved problems. The environment is called MetaEdit. MetaEdit is a flexible, graphical environment for methodology modelling. It is aimed to be a general metamodelling environment that can be interfaced with CASE shells[2] if necessary bridges are provided. Thus, it can be populated with several (meta)modelling approaches. It offers a platform to assess the strengths and weaknesses of various tool environments and allows for comparison and experimentation of their usefulness. Finally, it has a graphical interface to implement methodology modelling, and thereby it circumvents some of the weaknesses of the earlier environments.

The paper is organised as follows. In section 2 we outline some principles of metamodelling (methodology modelling) on which MetaEdit is built. In section 3 we introduce the design principles of MetaEdit and discuss its general architecture. Section 4 illustrates the

[2] By a CASE shell we mean a tool that can be customised by the users to support their own preferred methodologies.

internal structure and data model of MetaEdit which is the Object-Property-Role-Relationship model suggested by Welke (1988). Section 5 demonstrates how MetaEdit can be used to model a simple method such as Structured Analysis (De Marco 1978) data flow diagrams. The paper ends up with some comparisons to other work and outlines some future improvements in MetaEdit.

2. Modelling principles underlying MetaEdit

2.1. Modelling in systems development

In order to understand what in methodology modelling involves, we need an understanding of systems development process. We define information systems development as follows (Lyytinen *et al.* 1989, Lyytinen 1987, Welke 1983) :

> Information systems development is a *change process* occurring over time taken with respect to a set of *object systems* by a development group employing a collection of *methods* collectively referred to as a *methodology* to alter one or more object systems to realise or maintain one or more *objectives*.

The crucial concepts of the definition understanding are italicised. Hence, the definition

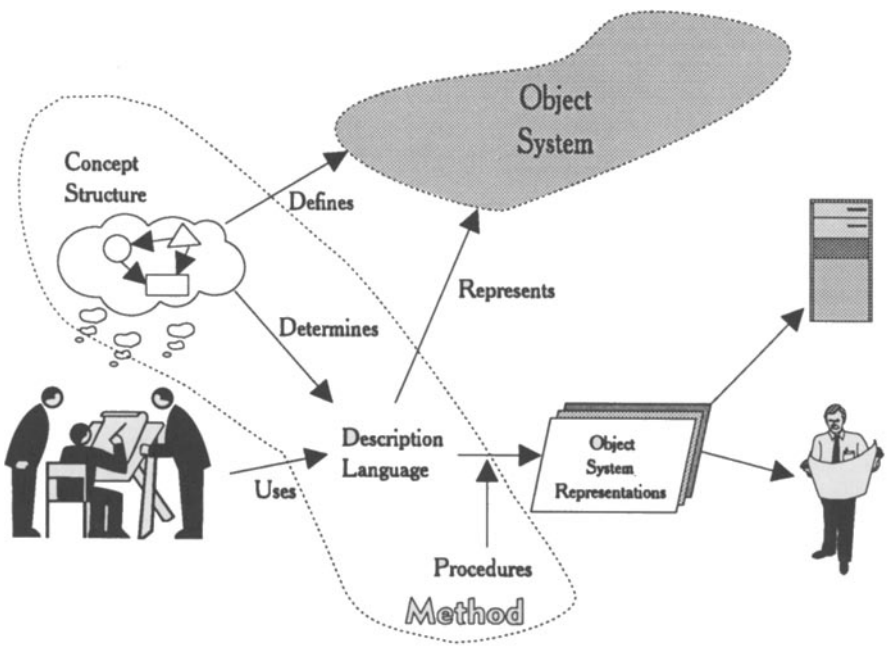

Figure 1. Object Systems in Systems Development

implies a set of other concepts as illustrated in figure 1. A development group's perception of object systems is enabled and constrained by a *concept structure* which determines what object systems are perceived by the development group. The concept structure is maintained and communicated by a language.

The language provides a means to convey the representations of object systems that are filtered from the reality using some concept structure by the development group. Usually a many to many relationship exists between the concept structures and languages. In the information systems literature terms like "description language" or "specification language" or just "notation" are usually used instead of our generic term "language". The language used may be natural language, but in many cases it includes also some formal language. The language is also graphical in many cases and deploys some diagrammatic notation.

A **method** is now a set of steps and a set of rules that define how a representation of an object system is derived/transformed using some concept structure and its language. A method thus embodies a set of concepts that determine *what* is perceived, a set of linguistic conventions and rules which govern how the perception is represented and communicated, and a set of procedural guidelines which state *in what order and how* the representations are derived/transformed. Usually the term "technique" is used for the description of the steps and associated derivation/transformation rules. The method may be well-defined and written or it may just be an outcome of habitualisation and evolutionary learning.

A **methodology** can now be defined as an organised collection of methods (a set of methods included in the methodology) and a set of rules which state by whom (roles), in what order (stage structure), and in what way (organising principles, quality criteria) the methods are used.

In **methodology modelling** we are interested in methodologies *qua* object systems that need to be perceived, described and changed by a *methodology development group*. Usually such methodology modelling task is done in order to change the systems development process in the organisation i.e. by trying to alter the set of methods, the set of roles, the set of stages and steps, or the set of organising principles and quality criteria associated with the development practices. One specific situation where such a task must be accomplished is when computer aided tools that support some of the methods in a methodology are introduced.

In this situation the methodology development group needs a concept structure and the associated language (metalanguage) to derive representations of the methodology under investigation. Such representations we call here *metamodels* or *methodology specifications* (cf. Brinkkemper 1990). Like in systems development the used metalanguage may be natural language or some formal language, and it may apply a graphical notation. Finally, this task of deriving methodology specifications can be computer supported or not computer supported. If some type of CASE tool is used to support

development methodology, this necessitates also computer support in deriving methodology specifications. Generally, this process is called Computer Aided Methodology Engineering (CAME) (Kumar and Welke 1988).

2.2. Motivation and modelling levels in MetaEdit

Historically, the languages to describe and analyse methodologies have been natural languages and the methodology specification process has been devoid of computer support. However, the introduction of computer supported tools has necessitated the application of formal and semi-formal metalanguages (see e.g. CRIS 1988) in methodology specification. Accordingly, these linguistic representations are used to guide the method application in a computer supported development environment (see Lyytinen *et al.* 1989). With the help of these languages one can clarify the concept structure underlying the method, and specify how it is tied with various notations (recall that the relationship here was many to many). In addition, one can specify the structural features of the method application as espoused by the methodology such as when, by whom, and in which order the methods are used. All this information can be then used to specify a methodology specific CASE tool for a given situation.

The goal of the MetaEdit is to provide a linguistic environment to develop and analyse methodology specifications, i.e. to produce a prototypical plattform for CAME. The specific feature of MetaEdit is that it provides a multilingual (several metalanguages can be used) environment in which methodology specifications can be developed, analysed and maintained.

To clarify how this is achieved consider figure 2. On the right hand side we can see three levels that are instrumental in delivering the functionality of a CASE shell. First, we have the level on which the IS developers work while they derive various object system representations about the IS under development. The syntax and the semantics of these representations are defined by a collection of metamodels (methodology specifications) that guide the use of the CASE tool. These metamodels in turn are founded on a modelling language and associated concept structure. This language/concept level is called here a meta-metamodel of the CASE shell. This level in fact offers the flexibility and extendability of the CASE shell environment.

Accordingly, these three levels are also present in MetaEdit as depicted on the left side in figure 2. Hence, in MetaEdit we distinguish between the meta-metamodel, the metamodel, and the model. However, these levels operate only one level "higher" than in the CASE

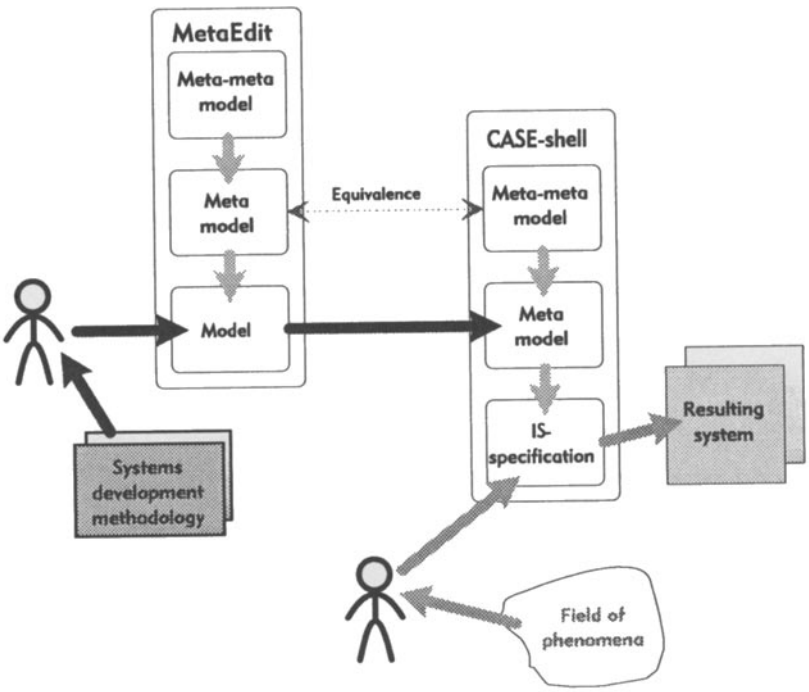

Figure 2. Three levels from meta-metamodel to model

shell[3]. Thus, the model level in MetaEdit defines the structure and functionality of the modelling language that will be used in representing models of IS i.e. the metamodel in the CASE shell. The metamodel in the MetaEdit is the model that is used to specify the methodology specification i.e. the meta-metamodel of the CASE shell. Multilingual specification fuctionality for methodology specifications is achieved in MetaEdit by offering a third level model-the meta-metamodel. On this level we can define the syntax and semantics of various metamodels that can be used to specify the methodology specifications. This flexibility of MetaEdit allows us to define the metamodels in the CASE shell in their "own language" if the meta-metamodel of the CASE shell has been defined in the meta-metamodel of MetaEdit. Typical "base languages" on the metamodel level of MetaEdit would be a set-oriented data model offered in RAMATIC (Bergsten et al. 1990), the E-R-model included into IBMs information repository concept, or the relational data model included into the IRDS (ISO 1989) proposal. This situation is depicted by the equivalence arrow in figure 2 between these two levels. By doing this we greatly simplify the subsequent automatic generation of the metamodel in MetaEdit that will used to guide the operations of the CASE-shell. In most cases this can be accomplished just by generating a textstream from the stored model in MetaEdit and doing some

[3] For this reason MetaEdit can also be used as a simple CASE shell.

syntactic opeations on the way. No complicated compilation between two different languages is needed.

Thus, MetaEdit is aimed to be used as a tool which offers a changeable methodology modelling language (metalanguage). This is achieved by defining a fixed set of rules included into MetaEdit's meta-metamodel. Next we shall illustrate how the three modelling levels on the left hand side of figure 2 are implemented in MetaEdit.

3. Design principles of MetaEdit

In this section, we shall introduce the basic design principles of MetaEdit. We begin with clarifying the concept structure and the associated language of a methodology model. Four domains are identified in the methodology and the mappings between them are defined. Next we explain the functions and architecture of MetaEdit and illustrate how this architecture reflects the selected metamodelling approach.

3.1. Four domains of model information

When specifying development methodologies, two aspects need to be separated: the conceptual content of a methodology, and its representation form. Accordingly, we shall distinguish two dimensions in methodology specification in which the contents of a metamodel can be located. These dimensions are called *type-instance dimension* and *conceptual-representational dimension* (see fig. 3).

In the **type-instance dimension** we a distinguish between the *types* that are included in the *metamodel* and their *instances* which make up the "things" observed in the modelling target i.e. the methodology under study. The type level determines what is allowed and legal

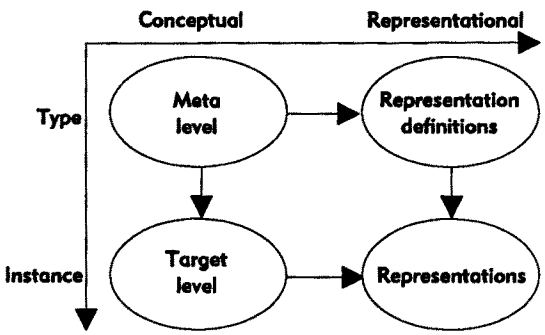

Figure 3. Four domains of model information in MetaEdit

in the instance level. The metamodel, with its generic representation definitions belong to the type level and the "things" in the model of the modelling target i.e. the methodology that user works with belong to the instance level. The same levelling principle is suggested in the IRDS framework (ISO 1989).

On the type level two types of type definitions can be distinguished: first, the conceptual type level items (*meta level*) specify the "things" (i.e. meta level types) possible in a metamodel. These meta level specifications include the properties of meta level types and their associations with other meta level types. In the second sort of type definition a meta level type is mapped to a representation definition.

When a MetaEdit user is building a model, s/he is working at the *target level*, i.e. the methodology level. The components of a conceptual model of the modelling target, the target level "things", are instances of the defined types on the meta level. Everything on the target level can be classified under the types on the meta level. Moreover, each conceptual target level "thing" may have multiple representations.

In the **conceptual-representational** dimension we observe the difference between concepts and their representations. This division between a conceptual domain and a representation domain has been widely applied since the ANSI/SPARC proposal of a three-level data base architecture (ANSI 1978). In MetaEdit, the types and their target level instances belong to the conceptual domain i.e. they do not carry any information about the way they are presented graphically. The graphical representation of a conceptual "thing" is defined by a contextual mapping from conceptual type definitions to their representational definitions.

Thus, for each type on the meta level, a *representation definition* is required. Representation definitions define the generic graphical behaviour of types. Consequentially, they capture the generic shape of a type, its line type (e.g. solid or dashed), and the textual labels included in the graphical symbol. For each target level instance we can suggest one or more *representations* that conform to the meta level definitions. These representations convey the place and size of the representation instance, and its connections to other representations.

The arrows in figure 3 should be read as follows: the existence of a "thing" in the domain where the arrow points to depends on the existence of "things" in the domain from which the arrow originates. A representation definition makes thus only sense if there exists a type on the meta level mapped to it. Correspondingly, a target level instance can exist only if it is associated with a type on the meta level. For each representation there must be a conceptual type on the target level and a representation definition. So, in principle there is a two-way road from the most abstract (meta level) to the most concrete (representation). The data structure behind all the functions of MetaEdit is designed according to this four-domain principle.

3.2. The functions and architecture of MetaEdit

In order to fully understand how metamodelling is carried out in MetaEdit, we shall explain its basic functions and associated architectural principles in more detail. Therefore, the general architecture of MetaEdit and associated user roles are briefly introduced in this subsection. The architecture, the functions and the user roles are depicted in figure 4.

As there were two levels of model information, the type and the instance levels, there are also two *user roles* associated with these levels: the tool manager and the methodology engineer. The tool manager is responsible for the type level specification bases, the metamodel base and the symbol base. S/he defines the modelling methodology to be used in metamodelling. The methodology engineer is the *real* user. S/he uses MetaEdit in specification work and is therefore responsible for the instance level specifications stored in the methodology specification base.

The third user role depicted in figure 3 is really a collection of roles. The most important of these is the role of a CASE tool implementor, who uses the output to customise the CASE shell to support the methodology that was previously modelled with the MetaEdit. The outputs of MetaEdit can also be useful to other people, e.g. the analysts (when a methodology manual is produced) or the tool manager (showing what methodology descriptions and metamodels are currently stored in MetaEdit).

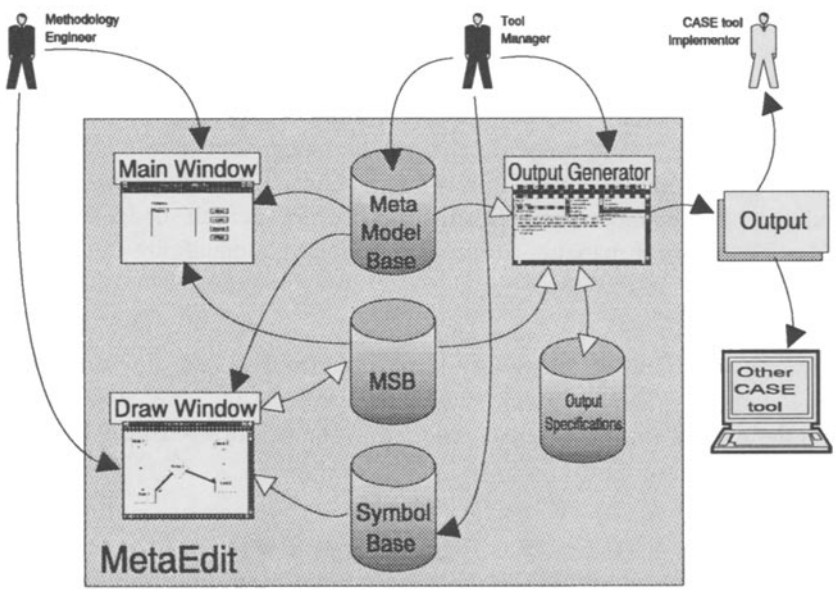

Figure 4. The Structure of MetaEdit

MetaEdit consists of three major functional areas (see fig. 4):

(1) **Main Window** offers the file and specification management functions. From the main window a methodology engineer can select the modelling methodology (from the metamodel base) when needed, and add, delete or change instance level specifications from the methodology specification base (MSB) s/he is working with. From the Main Window one can also call the other utilities of MetaEdit.

(2) **Draw Window** provides drawing functions to draw or edit specifications. It serves to input and edit these instance level specifications. Created instance level specifications are stored in the methodology specification base (MSB).

(3) **Output Generator** provides programmable utility functions that can create reports, generate code or retrieve data from the specifications in the methodology specification base. The tool manager (another user role) specifies the output specifications that are then stored in the output specification base. In our current work, the primary use of the output generator functions is to translate methodology specifications to formats needed in CASE-shells. A unique feature of this function is that it can also produce specifications to define MetaEdit's own metamodels. The output generator need also produce output in human readable formats for model validation and review.

The type level specifications belonging to the meta level are collected in the **metamodel base** in which the definitions of known metamodelling methodologies are stored. All window functions and utilities use a metamodel which determines their possible and legal actions in using a modelling methodology. When a methodology engineer starts MetaEdit, it immediately loads the default metamodel from the metamodel base which determines the default modelling methodology. If s/he wants to change the modelling methodology, s/he can load another metamodel (if defined) instead of the default model (which is now OPRR, see section 4). Tool manager defines the modelling methodologies that are stored in the meta model base.

The **methodology specification base** covers the instance level i.e. the target level and representations domains. The methodology specification base accumulates the target level "things" and their representations that are entered from the Draw Window by the methodology engineer.

The Draw Window uses the **symbol base** which defines how the meta level types behave graphically. This information consists of the generic representation definitions of the meta level types and they are made by the tool manager. It is possible to change the "look" of the current modelling language by changing the representation definitions in the symbol base. These changes to the symbol base do not affect the conceptual domains.

The key to the architecture of MetaEdit is the structure of metamodels in the metamodel base that determine both the functions of MetaEdit and the structure of the methodology specification base. In the following two sections we shall look closer to the generic structure of models in MetaEdit.

4. MetaEdit's meta-metamodel

In the design of MetaEdit we paid much attention to the flexibility of the tool. To obtain the best results, we made the metamodels of the tool to a very large extent customisable. A spin-off of this flexibility is that MetaEdit can be used for two somewhat different purposes. First, MetaEdit can be used as a simple, generic CASE tool with a flexible metamodel. Second, MetaEdit can be used as a methodology engineering tool (cf. Kumar and Welke 1988) which supports the building of metamodels to be loaded to other CASE shells. Now, two questions arise: 'what are the basic requirements of MetaEdit's meta-metamodels?', and 'what kinds of metamodels can be created using MetaEdit?'. In this section, the first question is dealt with. We shall introduce what kinds of "things" can be modelled using MetaEdit. The latter question is addressed in section 5.

In the first subsection we explain the structure of the fixed data model in which metamodels (methodology models) are defined. The contents of this data model are crucial in understanding the functioning of MetaEdit. In the second subsection we show how a specific metamodel is defined using OPRR as the methodology modelling approach.

4.1. The structure of the meta-metamodel

The choice of the meta-metamodel is critical in several ways. First, it defines what kinds of types can be included in the metamodel and so focuses the metamodeller's attention to certain constructs. Second, a meta-metamodel must have the sufficient semantic richness so that economic and powerful modelling principles are possible. In other words, the meta-metamodel must provide the sufficient semantic constructs to model a wide array of metalanguages with varying properties. On the other hand, meta-metamodel must be relatively simple, so that it can be learned and used easily.

In MetaEdit, the meta-metamodel is a fixed data structure based on the OPRR data model[4] (Welke 1988). We have chosen to use OPRR as the meta-metamodel for the following reasons:

---

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

(1) OPRR is a powerful data model. Several IS methodologies have been modelled successfully using OPRR. The fact that OPRR is used as a meta-metamodel in some commercial tools shows that (Meta Systems 1989).

(2) Although there are some restrictions (e.g. with decomposition), OPRR suits well to most graphical modelling methodologies. This will be shown in chapter 5.

(3) Yet OPRR is simple - there are only four basic concepts (meta types) in OPRR.

Thereby it seems to satisfy most of the requirements for the meta-meta model. The following definitions define the four basic OPRR concepts:

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 meta 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. An object always has a role in a relationship. This role defines what "part" an object plays in a relationship.

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

With the OPRR we adopt the following notation (cf. Welke 1988): an object type is presented by a rectangle, a property type by an ellipse, a role type by a circle, and a relationship type by a diamond. The name of each object, property, role or relationship type is written inside its symbol. The role type's connectivity restriction is also written inside its symbol. (For an example, see fig. 8.)

We have extended the OPRR model in MetaEdit by defining how OPRR concepts are mapped to their representations and how they are instantiated. Table 1 depicts these mappings.

An object type has always a symbol definition, and its instance is always represented by a *symbol*. A **symbol** is a graphical object that has one or more visible or invisible shape features to which *connectors* (i.e. lines) can be connected. A symbol may link to textual labels representing the properties of the object instance. Examples of symbols can be seen in figure 5. In MetaEdit, each target level object instance is presented by one or more symbols The graphical behaviour of the symbols is defined in the meta level presentation

OPRR concepts	*OPRR type representations*	*Concept instances*	*Representation of instances*
Meta level	**Representation definitions**	**Target level**	**Representations**
Object type	Symbol definition	Object instance	Symbol
Property type	Data type	Property value	Data field
Relationship type	Line type	Relationship instance	Connector
Role type	Symbol definition	Role instance	Terminal

Table 1. Mappings between domains

definitions.

A property value is presented in a *data field.* A **data field** is a slot into which users can enter data. A data field's form and its acceptable values are defined by a *data type.* A property may also be shown in the symbol's *label.*

A relationship instance is always presented by a line that is called a *connector.* A **connector** is a straight or jointed line between two objects. For each relationship type, its line type is defined. The attributes of a line type are e.g. thickness, colour, and solidity.

A role is also presented by a symbol. We shall call that symbol a *terminator.* A **terminator** is a symbol attached to a connector line and to another symbol representing an object instance so that the line ends at the terminator (e.g. an arrowhead). Terminators differ from object representations in that terminators do not have any "places" in which we can attach other connectors. A terminator is always placed to a connector's end. A terminator may also contain labels that convey the values of the role's properties. Terminators can also be left undefined, in which case the connector lines attach directly to the symbols. Examples of terminators are shown in figure 6.

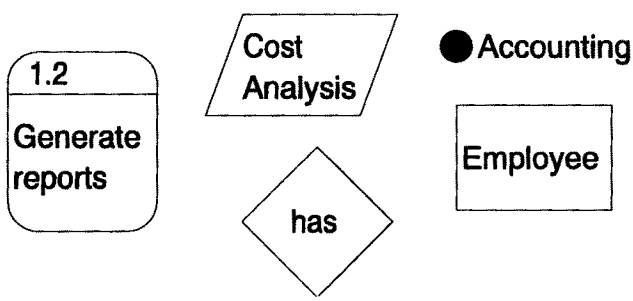

Figure 5. Examples of symbols

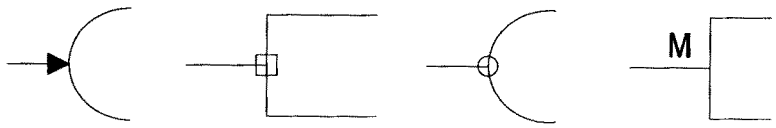

Figure 6. Examples of terminators

4.2. Defining OPRR in metalanguage

The next question is: how can we define and implement the meta-metamodel i.e. the fixed data model behind all the functions? To facilitate implementation, one solution is to develop still a higher level model and define a *meta metalanguage* to represent OPRR. In this work we can e.g. employ the Entity Relationship Attribute model. An EAR model of OPRR is depicted in figure 7. The figure follows the traditional rules of EAR-modelling (Chen, 1976): an entity is depicted by a rectangle, an attribute by an ellipse, and a relationship by a diamond. Cardinality constraints are shown beside the lines connecting the diamond to the rectangles.

The EAR model of the meta-metamodel of MetaEdit can be interpreted as follows. *Entities* are the "things" that exist on their own. *Relationships* can be interpreted as the syntax rules in the metalanguage: if two metatypes take part in the same relationship then

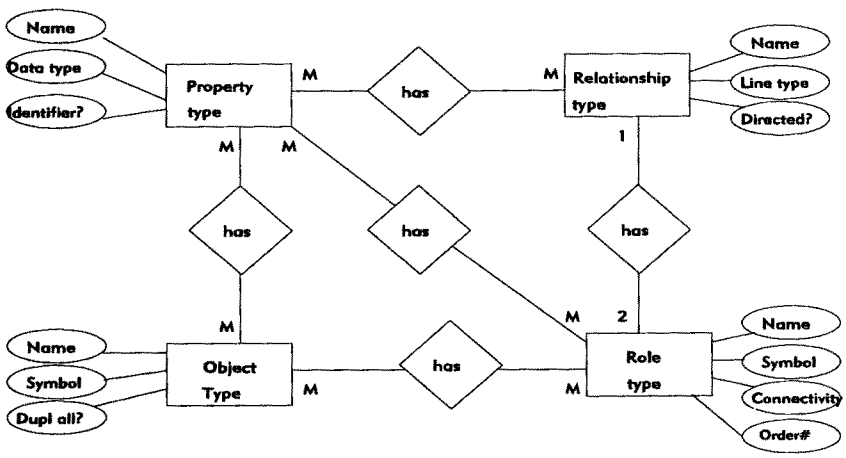

Figure 7. The meta-metamodel of MetaEdit represented in EAR

instances of these two metatypes (i.e. types) may have a link between them[5]. *Attributes* are variables that must be given values.

An `Object, Role` or `Relationship type` may be linked to several `Property types` and vice versa. An `Object type` must also be linked to one `Property type` through which its instances can be identified. An `Object type` may be linked to several `Role types` and vice versa. The only cardinality restriction in this model is that a `Relationship type` may be linked to only two `Role types`. This means that all `Relationship types` in MetaEdit's metamodels are binary. Because meta level `Relationship types` are represented as lines on the instance level (see chapter 5), it is difficult to graphically represent n-ary relationships consistently on the instance level.

In the meta-metamodel, every type is identified by a variable `Name` (attribute in figure). The values of `Name` must be unique. Also, each type has one variable that defines its generic representation. `Object type` and `Role type` have the variable `Symbol`, that includes the name of a generic symbol definition. A `Relationship type` has the variable `Line type` that defines the generic line type of its instances. Also `Property type` has the variable `Data type` that defines the data type of the value of a property instance in target level. Each entity in the model has also additional variables. For each `Object type, Dupl all?` defines if a target level object instance can have multiple representational instances. `Property type's Identifier?` defines if all the values of `Property type's` instances in target level must be unique. For a `Relationship type`, the variable `Directed?` defines if the `Relationship type` is directed. If it is directed, the target level objects in its roles must be given in the order of direction that is defined in `Role type's Order#`. Otherwise the order is free. `Role type's Connectivity` defines how many role instances of a given type an object instance can have.

This meta-metamodel is the basis for all definitions in MetaEdit. Depending on its instantiation, MetaEdit can be applied in different types of modelling approaches. Target level instances and their possible associations are fully determined by the definitions in the metamodel that can be changed on the user's request at any time.

4.3. OPRR presentation of OPRR

As mentioned in section 3, MetaEdit is used primarily to define modelling languages for other CASE tools and even for itself. To accomplish this, a metalanguage must be chosen

[5] And forward: if two types have a link between them, there may be two instances of the types in the target level that have a link between them.

and defined. When a modelling language is defined in a form of a metamodel, it is possible to use MetaEdit as a modelling tool to support the modelling methodology associated with that language.

If we want to use MetaEdit as a OPRR modelling tool, we must define the OPRR metamodel. Because the meta-metamodel of MetaEdit is also based on OPRR, we are obliged to represent OPRR by itself, i.e. we must develop an OPRR presentation of OPRR.

The OPRR presentation of OPRR is given in figure 8. The dashed rectangles in fig. 8 connect to every type a list of all the variables and values (properties), including their respective symbol definitions. These variables will not be explicitly shown in the representation of the object in MetaEdit.[6]

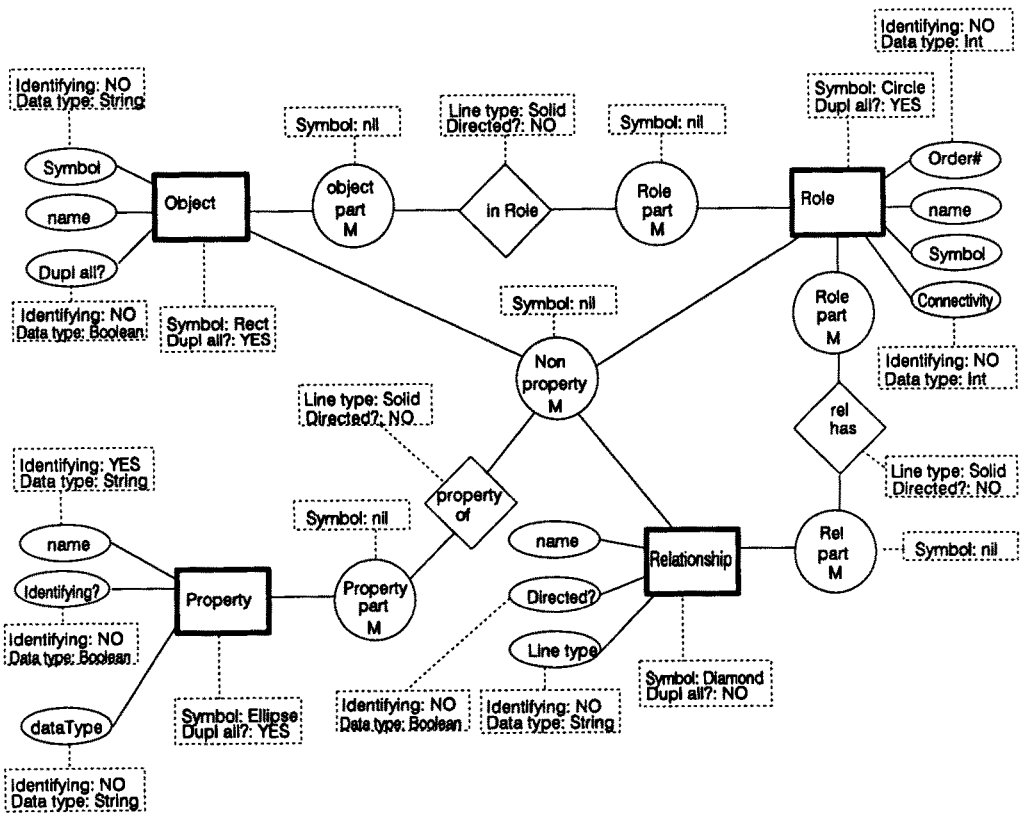

Figure 8. OPRR presentation of OPRR

The meta level object types are `Object`, `Property`, `Role`, and `Relationship`. In target level models, an object instance of type `Object`, `Role`, or `Relationship` can be connected as a `Nonproperty` to an object instance of a type `Property` (in a role `Property part`) through a relationship `property of`. An `Object` instance can be connected to a `Role` through a relationship instance `in Role`, and a `Role` instance to a `Relationship` instance by a relationship `rel has`. There are no symbols defined for roles because relationship lines between objects in our notation for OPRR are plain lines without any terminators.

The specification of OPRR in OPRR can be easily translated to a textual form. The resulting textual specification can then be read into MetaEdit and used as an entry in the metamodel base. When the metamodel for OPRR has been created, descriptions that are based on it can be read back to MetaEdit. In the same manner it is possible to model other methodologies and use them with MetaEdit, for example ERA-model would be easy to model in the same way.

5. How to derive methodology specifications in MetaEdit

Suggested approaches to methodology modelling deal primarily with specifying notations (e.g. Hekmatpour and Woodman 1987). Here, a methodology engineer is interested in notational problems: how to create and edit syntactically correct textual and graphical notations.

We believe that the notational focus is insufficient. When modelling information requirements of an organisation, analyst's attention is not focused on notation and its quality. Instead s/he is interested in the conceptual content of the requirements — i.e. in identifying objects and their relationships in the Universe of Discourse. When s/he has identified and named those objects of interest, s/he visualises them using a notation s/he is accustomed to. The visualised model may show that the analyst's thinking has been inconsistent or incomplete, and so serve as a means for validity and correctness checking. The notation is an aid, not the main focus in modelling. In the same vein, the primary interest in metamodelling should be in observing, deciding, and validating what kind of "things" and "facts" there are to be observed in systems development. Therefore the conceptual structure of the methodology should first be delineated.

---

[6] When working with MetaEdit, the properties will be entered and shown in a separate window associated to every object.

In this section we clarify how methodologies can be specified using MetaEdit and the developed OPRR metamodel. We shall begin with a conceptual specification by showing how to analyse and represent the conceptual structure underlying a methodology. Then we continue by specifying the generic representation of a method (specification language). This is exemplified by using a part of the structured analysis (SA) method (De Marco 1978) as the target method.

5.1. Conceptual specification of a methodology

Conceptual specification starts with resolving what kind of objects the methodology recognises, and what are the possible roles and relationships of these objects. Only after this problem has been settled, it is time to formulate the notation for the methodology.

Suppose we want to use some kind of data flow diagramming method in a CASE tool. In this method species, an information system is observed as consisting of interconnected processes (subsystems) that receive and send data. A process may receive data from other processes, from external sources, or from internal data stores, and send data to external entities (''sinks'') to other processes or to internal data stores. Moreover, an external entity and an internal data store can not be in a send-receive connection i.e. an external

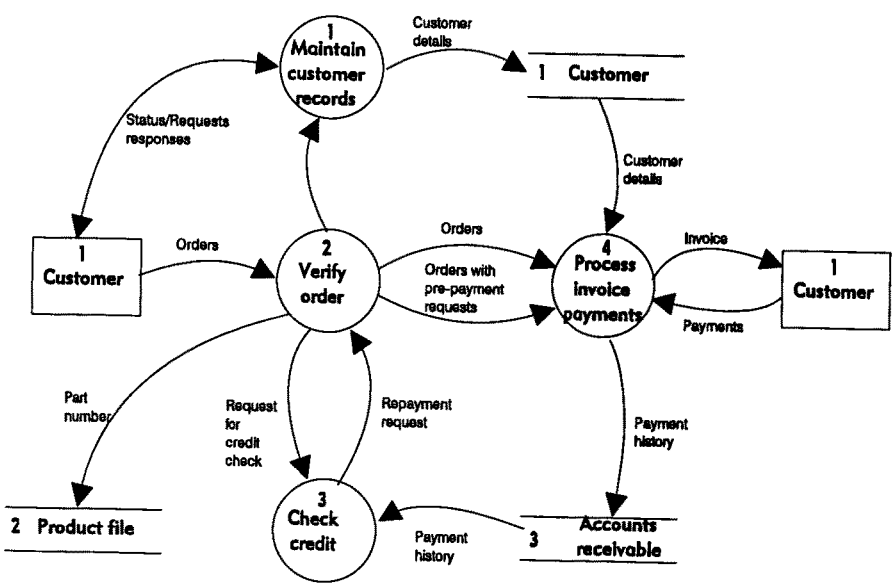

Figure 9. A data flow diagram

entity can not retrieve or manipulate internal data stores directly. So, we can distinguish three types of objects: processes, external entities, and data stores.

An example of a data flow diagram is shown in figure 9. The diagram consists of four processes ("Maintain customer records", "Verify order", "Process invoice payments", and "Check credit"), three data stores ("Product file", "Customer", and "Accounts receivable") and an external entity that is represented twice ("Customer").

In figure 10, the associated conceptual structure underlying the data-flow diagramming method is presented. The OPRR representation consists of three object types, `Process`, `Store`, and `External`, that are linked to each other by two relationship types `FFP` and `Flows`. Accordingly, a `Process` can be in the role type `From part` in the relationship type `FFP` (Flows From Process). This relationship type specifies a data-flow which begins from a `Process` and ends either to a `Process`, a `Store`, or an `External` (`To part` of the `FFP` relationship type). Thus, the role type `To part` defines instances of a *generalisation* in the sense that three different object types can be instantiated at this end of the relationship.

The second relationship type, `Flows`, represents a data-flow that begins from a `Store` or an `External` (`From part` of `Flows`). The only admitted destination of this data-flow is a `Process` (`To part`), because an external entity can not modify internal data stores.

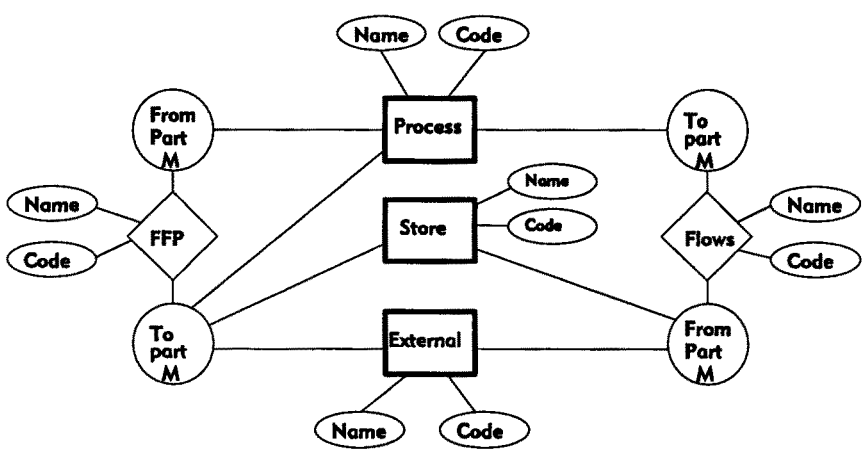

Figure 10. A OPRR specification of the data flow diagramming method

All the object types have the properties `Name` and `Code`. `Name` is a textual identifier of an object, and `Code` is an ordinal number associated with the object. Also the relationship types `FFP` and `Flows` have these properties. Other attributes could be defined if need be. For example, we could define the number of copies sent to the process, store or external in the FFP relationship.

5.2. Representation specification

The representation specification of a methodology should be consistent with conceptual specification. To avoid confusion and ambiguities, there should be a straightforward mapping from conceptual to representational specifications. For each object, property, relationship, and role type we should provide an unambiguous graphical definition.

We can thus extend the specification of the data flow diagramming method with its representation specification. In the following, one way of defining the generic representation of types is explained.

The representation of the *object types* is defined according to De Marco (1978).

- **Process** is represented by a symbol, circle, which has two textual labels, one in the upper region, and one in the middle of the symbol. The labels are associated to values of property types `Name` and `Code`.

- **Store** is represented by two parallel lines that have two textual labels between them. The labels are associated with properties `Name` and `Code`.

- **External** is represented by a rectangle that has also the same labels as the other object types.

The *relationship types* `FFP` and `Flows` are represented by a thin, solid line between associated symbols. The `To part` of both relationship types is represented by an arrowhead in the end of the relationship line (connector), whereas the representation of the `From part` is left undefined (no explicit graphical representation).

The data type of both `Name` and `Code` property types is *String*.

After formulating the representation specifications we have specified the data-flow diagramming method both conceptually and representationally.

6. Conclusions

In this paper we have introduced a metatool environment called MetaEdit which offers a flexible, graphical environment for methodology modelling. The environment is based on sound modelling and design principles that are based on the four domains of model information. Using these principles the OPRR-based fixed meta-metamodel was developed and the general architecture of MetaEdit was illustrated. Moreover, we showed how the meta-metamodel can be mapped onto the four domains. The applicability of MetaEdit was exemplified by using it as an OPRR modelling tool by which an OPRR presentation of OPRR was developed. Metaedit's use in methodology specification was demonstrated by deriving a simple methodology model of the data flow diagramming method.

Although MetaEdit embodies some new ideas, such as a graphical interface and an changeable metamodel, it has been influenced by several earlier research projects. The use of an extendible metamodel comes from the SEM environment (Teichroew *et al.* 1980) in which methodologies could be defined using a simple entity-relationship model. Another inspiration for the development of MetaEdit was Welke's extensions into ERA (1988), in which roles and their attributes were added to the metamodel definition. In the Metaview system (Sorenson *et al.* 1988) roles are also included in the metamodel definition. However, we are not aware of any approach in which the mapping between the conceptual metamodel and the graphical representation is so neat as in MetaEdit, i.e. none of the approaches above has considered the generic graphical representation of the meta types.

MetaEdit is now used to study methodology modelling. For example, we are currently defining RAMATIC's metamodel in MetaEdit, which will allow a graphical definition of RAMATIC's metamodels and the subsequent automatic generation of a large part of RAMATIC's methodology specifications. The definable metamodel also gives MetaEdit certain CASE tool characteristics and makes it usable for other kinds of data modelling purposes. For example, most of the prevailing data modelling methodologies could be metamodelled with in MetaEdit. The programmable output generator of MetaEdit can furthermore generate several types of output: methodology specifications, data base schemata, data definitions, etc.

Presently, the user interface of MetaEdit has been implemented in a prototype fashion. Current work is devoted to enhancing the functionality of MetaEdit by adding model storage and output generation functions to the tool. In the future, we shall pursue two goals: enhance MetaEdit's methodology modelling capabilities and add characteristics that give MetaEdit a customisable CASE tool functionality.

Acknowledgements

An earlier version of this report was presented in the first workshop on the Next Generation of CASE Tools. We are grateful for all comments that greatly benefited to improve the contents and form of the paper. Also prof. Welke's constructive comments have been helpful to arrive at the current version.

References

. ANSI,, "The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems," *Information Systems* **3** pp. 173-191 Pergamon Press, (1978).

. Bergsten, Per and Bubenko jr., Janis, Dahl, Roland, Gustafsson, Mats R., and Johansson, Lars-Åke, *RAMATIC - a CASE shell for implementation of specific CASE tools*, SISU, Stockholm (1989). First draft of a contribution to section 4.4 of the TEMPORA T6.1 report

. Brinkkemper, Sjaak, *Formalisation of Information Systems Modelling*, Thesis Publishers, Catholic University of Nijmegen, Nijmegen (1990). Ph.D. Dissertation

. Bubenko, jr., Janis A., *Selecting a strategy for computer-aided software engineering (CASE)*, SYSLAB University of Stockholm, Stockholm (June 1988).

. Chen, Minder, Nunamaker,jr., Jay F., and Weber, E. Sue, "Computer-Aided Software Engineering: Present Status and Future Directions," *Data Base* **20**(1) pp. 7-13 (Spring 1989).

. Chen, Minder and Nunamaker, jr., Jay F., "MetaPlex: an Integrated Environment for Organization and Information Systems Development," pp. 141-151 in *Procs. of the Tenth International Conference on Information Systems*, ed. Janice I. DeGross, John C. Henderson and Benn R. Konsynski,, Boston, MA (December 4-6, 1989).

. Chen, Peter Pin-Shan, "The entity-relationship model - toward a unified view of data," *ACM Transactions on Database Systems* **1**(1) pp. 9-36 (March 1976).

. Chikofsky, Elliot J., "Software Technology People Can Really Use," *IEEE Software*, pp. 8-10 (March 1988).

. CRIS88,, *Computerized Assistance During the Information Systems Life Cycle,* North-Holland, Amsterdam (1988). Proceedings of the IFIP WG 8.1 Working Conference on Computerized Assistance during the Information Systems Life Cycle CRIS 88

. De Marco, Tom, *Structured Analysis and System Specification,* Yourdon Press, New York (1978).

. Everest, Gordon C. and Alanis, Macedonio, *Selecting Computer-Aided Software Engineering Tools,* Dept. of Information and Decision Sciences, University of Minnesota (1989). An unpublished (?) research paper

. Harel, David, "On visual formalisms," *Communications of the ACM* **31**(5) pp. 514-530 (May 1988).

. Hekmatpour, S. and Woodman, M., "Formal specification of graphical notations and graphical software tools," pp. 297-305 in *ESEC '87: Proceedings of the 1st European Software Engineering Conference, Strasbourg, France, Sep 9-11, 1987 (Lecture Notes in Computer Science),* ed. H. Nichols and D. Simpson,Springer-Verlag, Berlin (1987).

. Index Technology,, *Customizer Reference Guide,* Index Technology Corporation, Cambridge, Ma (1987).

. ISDOS,, *An Introduction to the System Encyclopedia Manager,* ISDOS Project, Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, Michigan (September 1981). ISDOS Ref# 81 SEM-0338-1

. ISO,, *Information processing systems - Information Resource Dictionary System (IRDS) Framework,* ISO (1989). Draft International Standard

. Kumar, Kuldeep and Welke, Richard J., "Methodology Engineering: A Proposal for Situation Spesific Methodology Construction," in *Proceedings of CASE Studies 1988,* Meta Systems, Ann Arbor (1988). Meta Ref. #C8811

. LeQuesne, P. N., "Individual and Organisational Factors and the Design of IPSEs," *The Computer Journal* **31**(5) pp. 391-397 (1988).

. Lockemann, Peter C. and Mayr, Heinrich C., "Information System Design: Techniques and Software Support," pp. 617-634 in *Information Processing 86,* ed. H.-J. Kugler,North-Holland, Amsterdam (1986).

. Lyytinen, Kalle, "A Taxonomic Perspective of Information Systems Development: Thoretical Constructs and recommendations," pp. 3-41 in *Critical Issues in Information Systems Research*, ed. R. J. Boland Jr. and R. A. Hirschheim,John Wiley & Sons Ltd. (1987).

. Lyytinen, Kalle, *SYTI-Project: Research Plan*, University of Jyväskylä, Department of Computer Science, Jyväskylä, Finland (Spring 1988).

. Lyytinen, Kalle, Smolander, Kari, and Tahvanainen, Veli-Pekka, "Modelling CASE environments in Systems Development," in *Procs. of CASE89 The first Nordic Conference on Advanced Systems Engineering*, , Stockholm (1989).

. Meta Systems,, *QuickSpec Language Guide version 1.0*, Meta Systems, Ltd., Ann Arbor (January 1989).

. Orlikowski, W. J., "CASE Tools and the IS Workplace: Some Findings from Empirical Research," in *Procs. of the 1988 ACM SIGCPR Conference on the Management of Information Systems Personnel*, (April 7-8, 1988).

. Orlikowski, Wanda J., "Division among the Ranks: The Social Implications of CASE Tools for System Developers," pp. 199-210 in *Procs. of the Tenth International Conference on Information Systems*, ed. Janice I. DeGross, John C. Henderson and Benn R. Konsynski,, Boston, MA (December 4-6, 1989).

. Penedo, Maria H. and Riddle, William E., "Software Engineering Environment Architectures," *IEEE Transactions on Software Engineering* **14**(6) pp. 689-696 (June 1988).

. Siltanen, Aila, "The Impact of CASE Tools on IS Management," pp. 181-195 in *CASE on Trial*, ed. Kathy Spurr and Paul Layzell,John Wiley & Sons Ltd, Chichester (1990).

. Smolander, Kari, Tahvanainen, Veli-Pekka, and Lyytinen, Kalle, "How to Combine Tools and Methods in Practice - a field study," pp. 195-214 in *Advanced Information Systems Engineering*, ed. B.Steinholz, A.Solvberg, L.Bergman,Springer-Verlag, Berlin (1990).

. Sorenson, Paul G., Tremblay, Jean-Paul, and McAllister, Andrew J., "The Metaview System for Many Specification Environments," *IEEE Software*, pp. 30-38 (March 1988).

. Teichroew, Daniel, Macasovic, Petar, Hershey,III, Ernest A., and Yamamoto, Yuzo, "Application of the entity-relationship approach to information processing systems modeling," pp. 15-38 in *Entity-Relationship Approach to Systems Analysis and Design*, ed. P. P. Chen,North-Holland (1980).

. Venable, John R. and Truex,III, Duane P., "An Approach for Tool Integration in a CASE Environment," in *Proceedings of CASE Studies 1988*, Meta Systems, Ann Arbor (1988). Meta Ref. #C8812

. Welke, Richard J., "IS/DSS: DBMS support for information systems development," pp. 195-250 in *Data Base Management: Theory and Applications*, ed. C.W. Holsapple and A.B. Whinston,D. Reidel Publishing Company (1983).

. Welke, Richard J., *The CASE Repository: More than another database application,* Meta Systems, Ltd., Ann Arbor (1988).