Abstract

Current CASE tools support a limited number of methodologies. In contrast, every organization has adopted its own ways to develop information systems, limiting the applicability of the fixed methods that CASE tools support. To overcome this problem new methods are needed for defining and customizing methodologies for CASE tools.

A model by which methods can be modelled is introduced. It is called the OPRR datamodel (after Object-Property-Role-Relationship model). Also necessary graphical extensions that allow the definition of graphical modelling languages are introduced. An example which demonstrates how OPRR can be used in method modelling is shown. Finally, the shortcomings in the model are noted and some extensions to the basic model to remove these shortcomings are proposed.

* This research is funded by the Academy of Finland and the Finnish Center for Technical Development
1. Introduction

Despite the extensive interest in Computer-Aided Software Engineering (CASE), connections between systems development methods and CASE-tools are rarely discussed. The introduction of CASE to an organisation is understood as a straightforward solution to a technical problem, in which common analysis and design methods are taken into use with automated tools. Moreover, it is typically believed that the new technology will speed up systems development and produce higher quality systems that satisfy users better (e.g. Fairbairn, 1990). Little attention is paid on whether the methods that the tools support are suitable for the application area and the ongoing development practice.

However some research results (cf. Le Quesne, 1988, Smolander et al., 1990) show that the technology alone is not the solution to the systems development problems. For example, technology can not bring about changes that are beyond the systems development skills and knowledge in an organization. The systems development methods that an organization can use depend on the skills and knowledge, differing from an organization to an organization.

Still the methods support of CASE tools is usually fixed, leaving a minimal room for customization. Every organization has its own history and its methods that are shaped by prior experiences and trial and error learning. Hence, the organization's methods are not necessarily the same as the methods supported by the tools. In addition, different systems development problems require different methods of analysis and design. Different problems may therefore lead to the use of different CASE tools. As a result there would be no integrity between various pieces of design information.

A solution to the problems above would be to develop a systems development environment which could support the use of multiple methods concurrently. It would be based on a general datamodel which would help to map the features of systems development methods onto a “methodology specification”. In this type of environment, systems developers could use computer supported situation dependent methods, and still be able to communicate and transport design information between projects with different methodical needs.

Although there are some research prototypes of multiple methods environments coined as CASE-shells (Bubenko, 1988), they are not common yet. Some examples of such environments with definable methods are RAMATIC (Bergsten et al., 1989), MetaEdit (Smolander et al., 1991), and MetaView (Sorenson et al., 1988). Still, what is lacking is a well established theory of meta-models and multiple, concurrent methods. What would be the best or the most suitable model to define systems development methods and methodologies? Some proposals have been published (e.g. Teichroew et al., 1980,
and Welke, 1988, Sorenson et al., 1988, Wijers et al. 1990, Chen & Nunamaker, 1990) but there are only few practical applications and experiences of using these models.

The aim of this paper is to describe one possible data model for method modelling, the OPRR model (cf. Welke, 1988, Smolander et al. 1991) and its graphical extensions. The OPRR model is the basis for a set of commercial systems engineering tools. It is used in such commercial tools as IEW, Structured Architect Workbench, and QuickSpec (Forte, 1989). Also, the OPRR model is suggested as the basis for the CASE repository (e.g. Welke, 1988, 1989), the facility in which the systems specifications are collected.

However, the structure of OPRR is not yet specified accurately enough. The implementation of a CASE tool that is based on OPRR would need an exact specification of the model. Therefore we aim at clarifying the structure of OPRR and how to apply the model to method modelling. The version presented here (with extensions to graphical modelling) is the basis for MetaEdit, a graphical modelling tool with a definable meta-model (Smolander et al., 1991).

The structure of the paper is following: section 2 seeks the requirements for the ideal model: what are the general modelling requirements for method modelling? Section 3 describes the OPRR model and clarify to what extent it fulfils the requirements. The section ends at the examination of the strengths and limitations of the model.

2. REQUIREMENTS FOR METHODOLOGY MODELLING

Pertinent literature on CASE gives some suggestions on how systems development methods should be modelled for use with automated tools. Among the first proposals was the pioneering work on SEM (Teichroew et al., 1980), where the entity-relationship model was proposed as the basis for methodology models. At that time computing facilities were not mature enough to allow graphical modelling, and the need for extending the model to include graphical models was of little importance.

Teichroew et al's proposal was the starting point for the development of the OPRR model (Welke, 1988\textsuperscript{1}), in which the concept of role was used to extend the ER-model. The graphical extensions to

\textsuperscript{1} See also Sorenson et al., 1988 for a similar structure
OPRR were introduced in Smolander et al. (1991), when the OPRR model was used as a model basis in the development of the graphical editor MetaEdit. Other approaches suggested for method modelling include for example the modelling of graphical notations with set-theoretical formalisms in RAMATIC (Bergsten et al., 1989), and the application of NIAM (Nijssen & Halpin, 1989) and so called “task structures” to method modelling (Wijers et al., 1990).

The modelling of systems development methods requires certain properties from the modelling formalism. The following tentative list gives an overview of what must be taken into an account in an ideal method modelling environment:

1. Systems development methods are usually based on graphical notations. Also the recent developments in computing technology (powerful microcomputers, windowing systems) facilitate extensive usage of graphical modelling. Both the methods to be modelled and the technology require a possibility to model graphical notations of systems development methods.

2. So called repositories (e.g. Mercurio et al., 1990) will require a clear definition of concepts used in methods. From the CASE tool's point of view the method model should be consistent with the notation used, but from the repository's point of view the most important things are the actual concepts behind the method that often may have complex interactions with the notation. Therefore the modelling formalism should also facilitate the modelling of the associations between the concepts and the notation.

3. The systems analysts require that the tools are not too complex and difficult to use and understand. A powerful method modelling formalism will not lead anywhere if the results will not be understood by the users, or if the tool based on the formalism is too complex. The modelling formalism should provide tools to define user-friendly interfaces for CASE tools.

4. The nature of methods themselves require certain qualities from the modelling formalism (see e.g. Lyytinen et al., 1991). Many methods contain complex objects or decomposition that must be handled within the modelling formalism. The transformations between different phases of systems development and the consistency between multiple connected methods must be handled.

In Lyytinen et al. (1991) the modelling requirements for CASE are divided in three levels of complexity. The requirements according to Lyytinen et al. (1991) belong mostly to the fourth item above, although there are some necessary extensions to model organizational and communicational aspects of CASE tool use. In Lyytinen et al. (1991) the three different levels are:

1. Requirements to represent data related to one method use which include
the modelling of two dimensional representations, e.g. modelling of such languages as JSD, SA, and ER modelling.
Operational semantics over these representations (for example defining the allowed updates over these representations).
Three or n-dimensional representations (for example decomposition in data flow diagrams).
Representation of domain knowledge for validation e.g. to business models or application area.

2. Requirements to represent data related to multiple connected methods which involve the detection of
   Horizontal consistency, which means the consistency between different method descriptions on the same level of abstraction, for example between an E-R model and a data flow diagram.
   Vertical consistency, which is the consistency between semantically equivalent descriptions on different levels of abstraction, for example between E-R model and equivalent data base schema.
   Dynamic consistency, meaning the capability of keeping a trace of model changes. This can be compared to the version controlling problem.

3. Requirements related to describing the usage/users of the data that include for example
   organizational role models and associated subschemata for the repository
   communicational models
   method usage knowledge, for example selection guide-lines
   explanations of method use connections
   quality criteria for models

The two lists above bring forth a large set of requirements that can not be fulfilled at once. As far as we know, even the simplest part of requirements in Lyytinen et al. (1991), the requirements related to one method, has no established solutions in the context of the first set of requirements. Therefore we suggest that the development of method modelling formalism should take place gradually. First, a sound basis for the modelling of one method at a time should be created taking into account the first set of requirements above. After that the formalism should be augmented with possible extensions.

3. **MODELLING METHODS WITH OPRR**

In this section, the OPRR model is introduced and suggested as the basis for modelling one method at a time. In section 3.1. the static structure and the concepts of the OPRR are defined. The next section
shows how a valid model can be instantiated using a OPRR meta-model. The extensions to the OPRR model that facilitate the mapping between concepts and graphics are introduced in section 3.3. Finally, the strengths and weaknesses of the OPRR are evaluated in section 3.4.

3.1. The Concepts of the OPRR

The acronym OPRR comes from the words object, property, role, and relationship which are the meta-types in OPRR. Welke (1988, pp. 4-5) defines the meta-types in the following way:

*Object* is a “thing” which exists on its own. According to Welke, examples of objects in “systems development jargon” include process, flow, store, source, module, etc.

*Properties* are the describing/qualifying characteristics associated with the other meta-types. Typical properties include name, ID number, textual description, etc.

*Relationship* is an association between two or more objects. For example, there may be a relationship between a source and a process meaning that the process *uses* the source.

*Role* is the name given to the link between an object and its connection with a relationship. From the example above, the process would be the *user* and the source would be the *origin* of the data.

Formally, a meta-model can be defined in OPRR as a n-tuple \((P, R, X, r, p)\), where \(^2\)

\[O \quad \text{is a finite set of object types}
P \quad \text{is a finite set of property types}
R \quad \text{is a finite set of role types}
X \quad \text{is a finite set of relationship types}
r \quad \text{is a mapping}
\]

\[X \rightarrow \prod_{i=1}^{n}(R_i \times A_i), \text{ where } A_i = RV(R_i), \text{ RV: } R \rightarrow \Omega, \text{ and } n \geq 2.\]

In other words, \(r\) is a total mapping from the relationship types onto the cartesian product of cartesian products \(R_i \times A_i\), which defines the object types \(A_i\) that can be in

---

\(^2\) For an analogous definition, see Sorenson et al., 1988.
a role type $R_i$. The set of object types $A_i$ is returned from a total mapping $RV$ from the role types onto all subsets of object types. Because $n \geq 2$, the number of roles in a relationship is always at least two.

$p$ is a mapping $NP \rightarrow \text{Powerset}(P)$, where $NP = O \cup R \cup X$ is the set of non-property types. In other words, $NP$ is a partial mapping from the non-property types onto all subsets of property types. The mapping defines the property types associated with the non-property types.

Using this model we can capture the static information contents of a method. For a simple example we will take the class diagram from Object-Oriented Design (Booch, 1991). In Booch's modelling technique, there are two basic types of objects in the OPRR sense, classes and class utilities. According to Booch, a class is “a set of objects that share common structure and a common behaviour” and a class utility is “a collection of free subprograms” (Ibid., p. 513). There is an example of a class diagram in Figure 1, containing five classes and one class utility, “Controller Utilities”.

The example contains two types of relationships: a class may have an inheritance relationship with another class (as between “Heater” and “Actuator”), and a class or a class utility may use another class or class utility (as between “Environmental Controller” and “Heater”).

The visible property types in the example include Name for the classes and class utilities, and Cardinality for the use relationships. Booch also lists a set of other properties for classes and class utilities. From those we include here Visibility for classes and class utilities, and Cardinality for classes.
A partial OPRR structure of Booch's class diagram is presented in figure 2. In the figure, the OPRR notation is used: an object type is represented by a rectangle, a relationship type by a diamond, a role type by a circle, and a property type by an ellipse. A role type's symbol, circle, contains also the role type's connectivity restriction (see the next section).

The picture shows the conceptual structure of the simplified Booch class diagram. A class may be in the roles Descendant, Ancestor, User and Used. A class utility may be in the roles User and Used. The properties of a class are Name, Visibility and Cardinality, and of a class utility Name and Visibility. Both the roles User and Used have the property Cardinality, showing the number of use connections between classes and class utilities.

This conceptual structure can be represented in the form of sets as follows:

\[
\begin{align*}
O & = \{ \text{Class, ClassUtility} \} \\
P & = \{ \text{Name, Visibility, Cardinality} \} \\
R & = \{ \text{Descendant, Ancestor, User, Used} \} \\
X & = \{ \text{Inherits, Uses} \} \\
r & = \{ \langle \text{Inherits}, \langle \langle \text{Descendant, \{Class} \rangle, \langle \text{Ancestor, \{Class} \rangle \rangle \rangle, \\
\langle \text{Uses}, \langle \langle \text{User, \{Class, \text{ClassUtility}} \rangle, \langle \text{Used, \{Class, \text{ClassUtility}} \rangle \rangle \rangle \rangle \}\rangle \}
\end{align*}
\]

\[
\begin{align*}
p & = \{ \langle \text{Class, \{Name, Visibility, Cardinality}} \rangle, \\
\langle \text{ClassUtility, \{Name, Visibility}} \rangle \}
\end{align*}
\]
This kind of data structure is needed in the implementation of a OPRR based CASE environment. The set \( O \) contains the object types, \( P \) property types, \( R \) role types, and \( X \) relationship types, that are connected to each other with the mappings \( r \) and \( p \). In the next subsection we shall see how the instantiation, the creation of a specification according to the method model, happens.

### 3.2. The valid instantiation of a meta-model

What actions are allowed when a meta-model is specified using an OPRR model? The representation above defines the static information structure of a method: the object, property, role, and relationship types that are recognized. It is possible to derive the allowed actions using the static information structure as the basis. However, some additional information must be included.

The validity of the next four types of actions must be defined:

1. The creation of an object: what are the preconditions so that the creation would be legal?
2. The creation of a relationship which involves also the creation of the associated roles: what are its preconditions?
3. The deletion of an object: what are the side-effects of the deletion?
4. The deletion of a relationship: what are the side-effects of the deletion?

The change of an object or a relationship can be understood as a combination of a deletion and a creation.

#### 3.2.1. Creating objects

Each object to be created is an instance of an object type, and each object must have values for the property types that the object type is mapped with. An object may be presented as an ordered pair \( \langle t, pt \rangle \), where

\[
t \in O \text{ is the object type, and }
\]

\[
pt \text{ is a set of ordered pairs } \{ \langle P_1, v_1 \rangle, \ldots, \langle P_n, v_n \rangle \}, \text{ where } \{P_1, \ldots, P_n\} \subset P \text{ are the property types associated with the object type } t \text{ derived from the mapping } p: NP \rightarrow \text{Powerset}(P), \text{ and }
\]
\(v_1, \ldots, v_n\) are the values of the property types. From now on the ordered pair \(\langle t, p_t \rangle\) is called an object and the ordered pair \(\langle P_i, v_i \rangle\) a property.

To enable the control of consistency to the “real world”, the definition of property type must be extended. The properties must conform to certain constraints defined by the property type. The only pre-condition in object creation is that the following constraints for properties must be satisfied:

1. For each property type a data type is defined. The value of a property must belong to the set which is defined by the data type. The data type can be for example integer, character string, or even structured collection of data, picture, etc. This constraint will be called the data type constraint.

In the class diagram example (see fig. 1 & 2) the data types are following:

- Name is a character string
- Visibility is one item of the set \{“Imported”, “Exported”, “Private”, “Undefined”\}
- Cardinality is one item of the set \{0, 1, n\}

2. The property types may contain information about how the values may exist in association with the other values. The property values may be constrained so that only unique values are allowed for properties, i.e. no duplicate values are allowed among a certain property type. It may be stated that a value is a reference to a value among another property type, or that a value can not be null, etc. These constraints will be called integrity constraints (as in Tschiritsz & Lochovsky, 1982; van Griethuysen, 1982).

In Booch class diagrams (fig. 1 & 2) all the values of the property type Name must have unique values.

The objects in figure 1 may be listed as follows:

\[
\begin{align*}
\langle \text{Class}, \{ \langle \text{Name}, \text{“Environmental Controller”}\rangle, \langle \text{Visibility}, \text{“Exported”}\rangle, \langle \text{Cardinality}, 1 \rangle \} \rangle, \\
\langle \text{Class}, \{ \langle \text{Name}, \text{“Heater”}\rangle, \langle \text{Visibility}, \text{“Private”}\rangle, \langle \text{Cardinality}, 1 \rangle \} \rangle, \\
\langle \text{Class}, \{ \langle \text{Name}, \text{“Cooler”}\rangle, \langle \text{Visibility}, \text{“Private”}\rangle, \langle \text{Cardinality}, 1 \rangle \} \rangle, \\
\langle \text{Class}, \{ \langle \text{Name}, \text{“Lights”}\rangle, \langle \text{Visibility}, \text{“Private”}\rangle, \langle \text{Cardinality}, n \rangle \} \rangle, \\
\langle \text{Class}, \{ \langle \text{Name}, \text{“Actuator”}\rangle, \langle \text{Visibility}, \text{“Undefined”}\rangle, \langle \text{Cardinality}, \text{null} \rangle \} \rangle, \\
\langle \text{ClassUtility}, \{ \langle \text{Name}, \text{“Controller Utilities”}\rangle, \langle \text{Visibility}, \text{“Exported”}\rangle \} \rangle
\end{align*}
\]

3.2.2. Creating relationships

Similarly to objects, each relationship has a certain relationship type. However, the creation of a relationship involves also the creation of associated roles. A relationship can be presented by an ordered pair \(\langle x, r_x \rangle\), where
x is an ordered pair \( \langle t, p_t \rangle \), where \( t \) is the relationship type \( (t \in X) \) and \( p_t \) is the set of properties (see above);

\( r_X \) is a cartesian product of roles derived from the mapping \( r \), 
\[ r_X = \bigtimes_{i=1}^{n} r_i, \quad n \geq 2. \]

Each part of \( r_X \) is an ordered pair \( \langle \rho, o \rangle \), where
\( \rho \) is an ordered pair \( \langle t, p_t \rangle \), where \( t \) is the role type \( (t \in R) \) and \( p_t \) is the set of properties;
\( o \) is the object in the role (see “Creating objects”).

The preconditions for relationship creation are the following:

1. The data type constraints and the integrity constraints of the relationship and role properties must be satisfied (see “Creating objects”).

2. The connectivity constraint associated with each role must be satisfied. As in figure 2., for each role type a connectivity constraint is defined. The connectivity constraint is of the form \( \langle \text{min}, \text{max} \rangle \), where
   - \( \text{min} \) is the minimum number of roles of this type that an object must have.
   - \( \text{max} \) is the maximum number of roles of this type that an object may have.

For example, the connectivity of the role type Descendant in figure 2. \((0,1)\) means that a class may be a descendant only once, because the maximum connectivity is 1 (i.e. there is no multiple inheritance). The minimum connectivity 0 means that a class does not necessarily have to be in the role Descendant, i.e. it might be on the top of the class hierarchy. Because the connectivity of Ancestor is \((0, M)\), a class may be an ancestor to any number of classes (‘M’ meaning any number).

The relationships in figure 1. can be represented as follows, where the shorthand notation object(x) refers to an object whose property of a type Name has the value x (the notation applies only to this particular situation):

\[
\langle \langle \text{Uses}, \{ \} \rangle, \langle \langle \text{User}, \{ \{ \text{Cardinality}, \text{null} \} \} \rangle, \text{object(“Controller Utilities”)} \rangle, \\
\langle \langle \text{Used}, \{ \{ \text{Cardinality}, \text{null} \} \} \rangle, \text{object(“Environmental Controller”) \rangle} \rangle \\
\langle \langle \text{Uses}, \{ \} \rangle, \langle \langle \text{User}, \{ \{ \text{Cardinality}, 1 \} \} \rangle, \text{object(“Environmental Controller”) \rangle} \rangle, \\
\langle \langle \text{Used}, \{ \{ \text{Cardinality}, 1 \} \} \rangle, \text{object(“Heater”) \rangle} \rangle \\
\langle \langle \text{Uses}, \{ \} \rangle, \langle \langle \text{User}, \{ \{ \text{Cardinality}, 1 \} \} \rangle, \text{object(“Environmental Controller”) \rangle} \rangle, \\
\langle \langle \text{Used}, \{ \{ \text{Cardinality}, 1 \} \} \rangle, \text{object(“Cooler”) \rangle} \rangle \\
\langle \langle \text{Uses}, \{ \} \rangle, \langle \langle \text{User}, \{ \{ \text{Cardinality}, 1 \} \} \rangle, \text{object(“Environmental Controller”) \rangle} \rangle, \\
\langle \langle \text{Used}, \{ \{ \text{Cardinality}, n \} \} \rangle, \text{object(“Lights”) \rangle} \rangle \\
\langle \langle \text{Inherits}, \{ \} \rangle, \langle \langle \text{Descendant}, \{ \} \rangle, \text{object(“Heater”) \rangle} \rangle, \\
\langle \langle \text{Ancestor}, \{ \} \rangle, \text{object(“Actuator”) \rangle} \rangle \\
\]
As can be seen from the example, a relationship is composed of its properties (the sets of relationship properties in this example are empty), and of a set of roles, where each role is composed of its properties and of an object participating the relationship.

3.2.3. **Deleting objects and relationships**

A relationship can not exist without the objects in its roles. Therefore, when an object is deleted, also the relationships it participates must be deleted. For example, when the object “Heater” (fig. 1.) is deleted, also the use relationship to “Environmental Controller”, and the inheritance relationship to “Actuator” must be deleted. The deletion of a relationship implies also that all the roles of the relationship are deleted.

When deleting a *non-property* (an object, role, or relationship), the properties of the non-property must be deleted. On some occasions this might mean the violation of the integrity constraints. When doing a deletion of a property, the integrity constraints should be checked.

3.3. **The concepts and the graphics**

As noticed in Section 2, the possibilities to model graphical notations are needed when modelling systems development methods. On the other hand, information repositories require clear definition of concepts behind the methods. It is obvious that there is some kind of gap between the conceptual and representational (notational) specification of methods. The modelling of notation will need complex formalisms for defining spatial representation, whereas the modelling of concepts does not consider the spatial representations at all.

One possible solution to the division between conceptual and representational models would be to model the conceptual structure of a method and its notational structure independently and then define the mapping between the models. Unfortunately this could lead to confusion and inconsistencies. The mapping between concepts and notation could be ambiguous and the definition of the mapping would be extremely difficult job to accomplish.

In our approach, the dominant part of the meta-model is the conceptual structure. The representation is derived from the conceptual structure by making a representation definition for each object, property, role, and relationship type:
An object is represented graphically by a graphical symbol. For each object type, a symbol definition is included which contains the graphical shape and the places for labels. The labels represent the information derived from the property values.

A relationship is represented graphically by a line between two object symbols. For each relationship type, a definition of the line type is made. The definition includes for example the width, colour, and the pattern of the line.

A role is represented graphically by a symbol (for example an arrowhead) that is attached to the end of the relationship line. Each role type has a symbol definition similar to the object type. Naturally the size of the role symbol is much smaller than the size of the object symbol.

A property is represented in a data field or in a label of a symbol. The form of the data field is determined by the data type of the property type.

Although the conceptual structure is dominant, it does not mean that the representation definition does not state any requirements in practice. In MetaEdit (Smolander et al., 1991) which is based on the same model, we have extended the OPRR model so that it includes two characteristics that are not included in the basic model: the definition of the possibility to use representational duplicates and the definition of the relationship direction. Both definitions are made to enhance the representational specification capabilities.

The possibility to present duplicate objects is needed for the sake of clarity. For example, in OPRR diagrams (e.g. fig. 2) a property (“Name”, “Cardinality”, ...) may be represented in many locations at a same time. Without the possibility to create duplicates, the graphical representation would be quite a mess, the lines crossing between objects like a spider web. Therefore this possibility is needed.

When creating duplicates an integrity constraint needs to be checked: if one representational instance of an object is modified, the modification should apply to all representational instances of the same object. This obliges to a clear distinction between the concepts and the graphical representation. The “information” lies on a “conceptual level” while the representation mirrors the concepts.

The definition of the relationship direction deals with the selection of an appropriate relationship and role types when drawing a relationship line. The direction defines the order of roles: what is the order in which the role types are selected? For example, when creating an inheritance relationship between two classes (see fig. 2) the order of roles may be defined to be (Descendant, Ancestor), i.e. the class in a role “Descendant” is chosen before the class in a role “Ancestor”.

96
3.4. LIMITATIONS

OPRR is not the perfect and final model for method modelling. It meets only some of the requirements listed in section 2. It is a model which neatly combines the conceptual and representational parts of method modelling, but leaves many open questions. Usually the practice shows the greatest weaknesses of a theory. In our work, especially in the development of the graphical tool MetaEdit, the experiments has revealed us some shortcomings of the OPRR model. Some of them are mentioned in the following.

The weakest point of OPRR is its flat structure. It does not give any possibilities to model naturally 3- or n-dimensional structures nor complex objects (c.f. Harel, 1988; Neuhold, 1986). An example of a complex object can be found from figure 3, which is an object diagram from Object-Oriented Design (Booch, 1991). An object “theGrowingPlans” contains a set of objects, growing plans. “theGrowingPlans” is a complex object that is composed of a set of other objects. Typically data-flow diagrams (e.g. Gane & Sarson, 1979) contain complex objects (decomposition). A process in a data-flow diagram may be composed of a set of processes and data stores that are represented in another diagram. In (Lyytinen et al., 1991) this is called a three-dimensional representation. This problem may be resolved by taking new meta-concepts in to use. The concept of set or aggregate is needed for modelling of this kind of behaviour. The OPRR model must be extended to include set or aggregate as a modelling concept.

Another shortcoming of the OPRR model is that it does not contain any concepts for defining the connections of multiple connected methods, e.g. between different phases of systems development. Such definition would need a concept of modeltype (c.f. Bergsten et al.,1989) or transformation to be taken into use.
Also the graphical modelling capabilities should be extended. The way described above is sufficient for simple notations, but there are many methods whose notations can not be modelled using the OPRR approach. For example the arrow in Figure 3 between “aGrowingPlanController” and “anEnvironmentController” meaning that the messages “SetTemperature” and “SetLights” are synchronous can not be represented using the OPRR graphical extensions. The representation would need that there were conditional symbols given to properties. The synchronization is a property of the message-relationship between objects. When the synchronization gets a value “synchronous”, the arrow is drawn beside the relationship line. When the synchronization gets other values, an appropriate symbol is drawn. The symbols $\exists$ and $\forall$ are just symbols for role types “shared field” and “field”.

4. **Conclusions**

In this paper, the OPRR model with graphical extensions was suggested as a basis for method models. However, as explained in section 3.4., the model is not yet powerful enough. The flatness of the model and its inability to present complex objects were discovered to be greatest limitations. Also the graphical modelling capabilities need to be extended.

Despite its weaknesses, the OPRR model has shown its usability in practice. With the extensions mentioned earlier, the concept of set, modeltype and the graphical definitions for property types, the OPRR model would serve well as a data model for method modelling. Because the model is relatively simple, the production of outputs from an OPRR based model would also be relatively easy. The development of SQL-like query and reporting language would for example facilitate customized generation of code from models made with suitable methods. Some steps to the direction of code generation has been made with MetaEdit.

This paper has shown how OPRR can capture the static information contents of a method. In addition to the information contents, a complete CASE shell with method modelling capabilities should also include possibilities to model the dynamic aspects of systems development. An open question is how to model the design transactions and how to cope with incomplete design information. Also the modelling of spatial behaviour of the representation is a problem that can not be easily solved. The next generation of method modelling should give a solution to these problems.
ACKNOWLEDGEMENTS

Thanks to Mauri Leppänen and Kalle Lyytinen for their helpful comments on the first version of this paper.

REFERENCES

Bergsten et al., 1989

Booch, 1991

Bubenko, 1988

Chen & Nunamaker, 1989

Fairbairn, 1990

Forte, 1989

Gane & Sarson, 1979

Harel, 1988

Le Quesne, 1988

Lyytinen et al., 1991
Mercurio et al., 1990

Neuhold, 1986

Nijssen & Halpin, 1989

Smolander et al., 1990

Smolander et al., 1991

Sorenson et al., 1988

Teichroew et al., 1980

Tsichritzis & Lochovsky, 1982

van Griethuysen, 1982

Welke, 1988

Welke, 1989

Wijers et al., 1990