14

Comparison of four Method Engineering languages

F. Harmsen[1], M. Saeki[2]

[1]University of Twente, Department of Computer Science
IS Design Methodology Group
P.O. Box 217, 7500 AE Enschede, Netherlands

[2]Tokyo Institute of Technology
Department of Computer Science
2-12-1 O-okayama, Meguro-ku, Tokyo, 152, Japan

Abstract

Currently, several languages to represent and manipulate parts of IS engineering methods, techniques and tools are being used. These so-called Method Engineering languages can be classified into four categories: product-oriented, object-oriented, process- and decision-oriented, and hybrid. In this paper representatives of each of these categories are being reviewed. Meta-models of the languages are given, and each description is illustrated by an example specification. Focus of comparison is expressive power. The Method Engineering languages are compared on the basis of a number of requirements, which are deduced from the notions used in the Method Engineering domain.

Keywords

Method specification languages, Comparisons

1 INTRODUCTION

Method Engineering is the discipline to construct new methods from parts of existing methods. In order to successfully apply Method Engineering principles (Kumar and Welke, 1992; van Slooten and Brinkkemper, 1993), a specification language is needed with which method fragments, i.e. components of IS development methods (Harmsen et al., 1994), can be described and manipulated. We define a *Method Engineering language* as a modelling technique with the purposes:

- to represent IS development and management methods and fragments thereof, and
- to enable the assembly of information systems development methods by offering constructs to manipulate method fragments.

S. Brinkkemper et al. (eds.), *Method Engineering*
© Springer Science+Business Media Dordrecht 1996

If such a language is only able to represent method fragments, we call it a meta-modelling language. Brinkkemper claims that every conceptual modelling language is suitable to serve as a meta-modelling language (Brinkkemper, 1990), and therefore suitable to represent method fragments. Various meta-modelling applications of languages originally intended for other domains, such as LOTOS in software process modelling (Saeki et al., 1991), show the validity of this claim. However, the expressive power with respect to Method Engineering of the various conceptual modelling languages differ considerably, turning some modelling languages more suitable for method specification and manipulation than others. Moreover, Method Engineering projects have different goals and objectives, which, again, makes it hard to choose a language superior in all cases.

As Method Engineering is becoming more mature, different schools of thought have been established concerning representation and manipulation languages. One school adopts a data-oriented approach, stressing the representation of the product aspect of methods. Method Engineering languages of this type include (G)OPRR (Smolander, 1992; Kelly et al., 1996), PSM-LISA/D (ter Hofstede, 1993), NIAM Concept Structure Diagrams (Brinkkemper, 1990; Wijers, 1991), semantic data models (Sowa and Zachman, 1992), and ASDM (Heym and Österle, 1992). Others adopt an object-oriented approach, such as Telos (Mylopoulos et al, 1990), Metaview (Sorenson et al., 1988), and Object Z (Saeki and Wen-Yin, 1994). The third school consists of languages evolved from or direct towards software process modelling and capturing design rationale. Among these languages are Task Structure Diagrams (Wijers, 1991; Verhoef and ter Hofstede, 1995), HFSP (Katayama, 1989; Song and Osterweil, 1992), ALF (Benali et al., 1989), and MERLIN (Emmerich et al., 1991). A last category are the so-called hybrid languages, taking into account different aspects and offering often explicit operations for Method Engineering. MEL (Harmsen and Brinkkemper, 1995ab) is an example of this type of language.

In this work, we have taken from each category one representative. We compare Object-Z, MEL, GOPRR, and HFSP. We also review the data models (meta-models) of these four Method Engineering languages. For the comparison, we have listed a number of requirements, some of which are quite generic, applying to all conceptual modelling languages, whereas others only address Method Engineering.

Related research is performed in the area of comparison of methods and techniques and CASE tools. Hong et al. (1993) compare eight object-oriented methods by comparing their data models. A "super method" acts as a reference model for the methods compared. Iivari (1994) relies, in line with the "CRIS" approach (Olle et al., 1983, 1991), on a normative comparison of object oriented methods, as he draws up a number of requirements which are compared with the methods' properties. In contrast to this, Oei and Falkenberg (1994) define a set of basic transformations, called the Meta Model Hierarchy transformations, to transform method fragments, via a sequence of basic steps, into each other. Operations are associated with the quality attributes expressive power, genericity and liberality. Their main criticism on an approach using a reference technique is, that it would be easy to create yet another modelling technique, whose concepts are not or only partly covered by the framework. Song and Osterweil (1992) take a similar approach as Hong's, comparing different methods on the basis of their underlying models. However, Song and Osterweil concentrate on process models, which are represented in the process modelling language HFSP. Other related research focuses on normative comparison of conceptual data models (Venable, 1993), CASE tools (Wijers and van Dort, 1990) or meta-CASE tools (Marttiin et al., 1993).

We have adopted the traditional CRIS approach for comparing Method Engineering languages. We claim that in limited application domains, such as Method Engineering, normative comparisons are possible. Concepts and relationships are fairly stable, in contrast to general application domains which are continually subject to additions. In this paper we have investigated the Method Engineering domain, and translated the results of these investigations to requirements for Method Engineering languages.

This paper is structured as follows. Section 2 deals with the requirements that are to be imposed on Method Engineering languages. Section 3 reviews the four chosen languages, whereas in section 4 a comparison between these languages based on the requirements is made. The paper ends with conclusions and suggestions for further research.

2 REQUIREMENTS FOR METHOD ENGINEERING LANGUAGES

According to Oei and Falkenberg (1994), a conceptual modelling language should have the expressive power to model the application domain in an effective manner, and should be practical to apply with respect to convenience, efficiency, and learnability. The application domain is Method Engineering, in particular representation of methods. In general, a Method Engineering language should also enable the administration of method fragments in the method base, the selection of method fragments, and their assembly into a situational method, but these operational aspects will not be considered in this paper.

2.1 The Method Engineering domain

A Method Engineering Language should support representation and manipulation of all types of method fragments. Not only should, for instance, a complete method like OMT be represented, but also its elementary concepts, such as "Object", "Activity", and "State". Moreover, specification of both products and processes should be supported. We have developed a classification framework which clarifies the different types of method fragments. We classify method fragments along the three dimensions *perspective, abstraction*, and *granularity layer*.

The perspective dimension constitutes of the *product* perspective and the *process* perspective. Product fragments are deliverables, milestone documents, models, diagrams, or concepts. For instance, "Functional Specification" and "Data Flow Diagram" are product fragments. Process fragments represent the stages, activities and tasks to be carried out. Examples of process fragments are: "Create Data Flow Diagram", "Perform Requirements Analysis", or "Make Data Model".

Furthermore, we distinguish between *conceptual* method fragments and *technical* method fragments. Conceptual method fragments are objective descriptions of information systems development methods or part thereof. For instance, a set of guidelines in the Information Engineering book (Martin, 1990) to construct ERD's is a conceptual method fragment, as is the description of ERD's concepts and relationships. Technical method fragments are the operational parts of a method, i.e. the tool components. An Entity Relationship Diagram editor is an example of a technical method fragment, as is its associated repository or the hypertext ERD procedure in a CASE tool process manager. Some conceptual fragments are to be supported by tools, and must therefore be accompanied by corresponding technical fragments.

A method fragment can reside at one of five possible granularity layers:

- Method, which addresses the entire object system. For instance, the Information Engineering method resides at this granularity layer.
- Stage, which addresses an abstraction level of the object system. An example of a method fragment at the Stage layer is a Technical Design Report.
- Model, which addresses an aspect of an abstraction level. Examples of method fragments at this layer are the Data Model, and the User Interface Model.
- Diagram, addressing the representation of an aspect of an abstraction level. For instance, the Entity Relationship Diagram or the State Transition Diagram are at this granularity layer.
- Concept, which addresses the concepts and associations the method fragments on the Diagram layer, as well as the manipulations defined on them. Examples are: "Entity", "Entity is involved in Relationship", and "Identify entities".

2.2 Requirements

Based on the framework presented in section 2.1, the expressive power and practicality requirements have been adapted for Method Engineering, resulting in a number of requirements for Method Engineering languages. The requirements can be viewed from three perspectives: *importance*, *genericity*, and *Method Engineering domain*. The perspective *importance* is introduced because some requirements have more impact on modelling systems development methods than others, or are a logical consequence of others. The *genericity* perspective focuses on the extent to which requirements are generic, or Method Engineering specific. The *Method Engineering domain* perspective encompasses the three dimensions introduced in section 2.1.

Prerequisite

To our opinion the most important requirement, and therefore a prerequisite to all other requirements, is *suitability*. This requirement implies that the Method Engineering language should be learnable, efficient, and convenient for the method engineer. The language should contain concepts and constructs corresponding with the method engineer's intuition and the Method Engineering domain. The products and processes are to be modelled in an efficient way.

Method Engineering Domain

A group of requirements of secondary priority, all of equal importance and addressing the Method Engineering domain, are:

- Support of representation of both method *processes* and method *products*. The functional and behavioural contents of each activity need to be represented. The components and structure of products need to be specified. Also, support for hierarchical decomposition of both process descriptions and product descriptions to enable specification of contents should be provided. For instance, OMT's Object Design activity, a process description, consists of activities like "Optimise access paths", "Adjust structures", and "Design attribute details". OMT's Object Model, a product description, consists of concepts like "Object" and "Class", their properties, and relationships between them. The Method Engineering language should provide notations and means to represent these activities and products.

- Support of representation of both *operational* (*development*) and *project management* aspects of methods. These aspects should be distinguishable, but relationships between them have to be identifiable. For instance, a "Plan Object modelling" activity is related to the Object modelling activity (which is planned by it), its sub-activities, and its products, although it addresses a different aspect of the method.
- Support of representation of the *conceptual* and the *technical* (i.e., tools) side of methods. As with the difference between development and project management, conceptual aspects and technical aspects are related, but should also be distinguishable. A conceptual description of Entity Relationship Diagrams, for instance, only contains the concepts "Entity", "Relationship", "Attribute", and so forth. Properties are "name" and "definition". An implementation of Entity Relationship Diagrams, for instance a diagram editor or part of a CASE tool repository, needs additional concepts and properties, such as specifications of the symbols with which concepts are represented (rectangle, diamond, circle), access paths to internal storage, etc.
- The ability to *formally express constraints and rules* concerning method fragments. The mere representation of process fragments and product fragments does not suffice. Relationships among method fragments are subject to constraints, and method fragments themselves behave according to rules. For instance, a rule is: each data flow in a DFD should be specified by an Entity Relationship Diagram. Such rules cannot be expressed by simple cardinality constraints, but are important for the consistent application of method fragments. In particular for effective tool support, rules should be formally expressed. It is necessary that a Method Engineering language provides support for such formalised constraints and rules.
- Support for representation of *actors* and *roles* in the systems development process. In order to assign responsibilities and duties to people (the key factor) involved in systems development, it is necessary to provide mechanisms for representing so-called actors and the roles they play. An actor is considered an actual person, such as Henk de Vries or Haruhiko Suzuki, which play roles like project manager, analyst, programmer, and so forth.
- Distinction between *instance level* and *meta-level*, and support of both of them. The instance level consists of the actual systems development processes, such as "Create Inventory control object model", and products, such as "Inventory control object model". The meta-level addresses the method to be used on the instance level. Method fragments, such as "Create object model", belong to this level. Actual development processes, including their products and tools, are therefore instances of method fragments. The distinction is important, as developers generally only deal with the instance level, which is the level where the actual job is done. The meta-level is prescriptive, and influenced by the instance level, because experience gained in the actual development process has to be captured by the method fragments.
- Support of *non-determinism*. Actual systems development generally takes not place according to strict sequences of processes. Usually, a lot of non-deterministic activities take place, for instance in joint application development or prototyping, for which no strict sequencing can be provided. Non-determinism is also used to handle exceptional or unforeseen cases, such as excessively running out of time.
- Support of *parallel* processes. Systems development is teamwork, which is reflected in the parallel nature of many method processes. A Method Engineering language should be able to support this parallelism, both on the meta-level, for instance the parallel execution of data modelling and process modelling activities, and on the instance level, e.g., the parallel execution of data modelling activities A and B.

- Support for recording *design rationale*. As was already noted before, the instance level constantly forces to make changes on the meta-level. Project experience should be accumulated in the method fragments, which are relatively dynamic notions. One of the main mechanisms to benefit from experiences made earlier, is to record design or decision rationale. This implies that not only the decisions are described, but also the reasons why decisions have been made, including their pro and con arguments, motivations, etc. Design rationale can be viewed both on the instance level, for instance the pro and con's of introducing an additional object class "Level" in an inventory control system specification, and on the meta-level, for instance the reasons why State Transition Diagrams have been chosen in a particular method.

Generic requirements

Besides the overall prerequisite of suitability, and requirements regarding the Method Engineering domain, some requirements for Method Engineering languages are applicable to any specification language. These generic requirements, which are of less priority than the other two groups, are:

- Support of *modularisation* of method fragments. Effective re-use of method fragments demands for a modularisation mechanism in the language. Method fragments are essentially black boxes, which are to be selected and assembled based on their external specification; internal details should be encapsulated as much as possible. Related to this is the need for effective characterisation of method fragments. A language should supply the means to valuate properties in such a way, that each method fragment is uniquely identifiable. For instance, an Entity Relationship Diagram can be characterised by a goal (such as: "Data modelling"), a maturity level, a description of the capabilities needed to apply the technique, a set of application domain descriptions, and so forth. A method engineer does not need to know which concepts and relationships play a role in ERD, but can select by investigating its properties. The properties also provide a fine-grained classification of method fragments to support modularisation, as opposed to the coarse-grained classification described in section 2.1.
- Support for defining *views*. A method comprises everything needed for performing systems development, be it for a general case or for a specific case, resulting in a huge amount of activity descriptions, product descriptions, tools, etc. However, each actor in the systems development process requires only a relatively small portion of the method, which is its view on the method. For instance, a project manager has a different view, including project management activities, products and tools, than a programmer, who needs other parts of the method. The definition of views reduces complexity for a single actor, and increases learnability.
- Unambiguity, implying that a Method Engineering language should be formally, mathematically defined to avoid multiple interpretations and to be able to enact the method.

Venable (1993) uses a similar classification for evaluation of conceptual data models. He distinguishes between *criteria for semantic concepts*, *criteria for syntactic constructs*, and *criteria for relationships to other areas*. The first group contains criteria such as *richness* (expressive power), *minimality* (suitability, in particular efficiency), and problem domain correspondence (all the requirements concerning the Method Engineering domain). The criteria for syntactic constructs relate to the graphical representation of concepts and the relationship between syntax and semantics. The third group roughly corresponds to the *generic* requirements described above.

3 REVIEW OF METHOD ENGINEERING LANGUAGES

In this section four representative Method Engineering languages are reviewed. The concepts and relationships of each language are described in a data model or *meta model* (Brinkkemper, 1990). Meta models are depicted in an Entity Relationship Diagram notation, consisting of entities and relationships. This notation is used for the sake of presentation, to provide a quick overview. To get a further impression of each language, a small example concerning Entity Relationship Diagramming has been conceived.

3.1 Object-Z

The formal specification language Object-Z (Duke et al., 1991; Saeki and Wen-Yin, 1994) is an object oriented extension of the Z language semantically based on ZF set theory. In the object oriented paradigm, the system to be specified is considered as a collection of individual objects having internal states. Object-Z defines the objects by using class concepts where the definitions of their states (state variables), initial states, and the operations related to them are encapsulated. The class schema for the specification of a class may contain schema's for defining operations permitted on the objects. A class can inherit states and operations from other classes.

Figure 1 depicts the meta-model of Object-Z. In this figure, method fragments are modelled from the product and the process perspective. From the product perspective, the structures or types of the produced products and constraints on the product components are specified. To specify the processes, permitted manipulations on the method products are defined as operations. Behavioural constraints, such as execution ordering, are specified as pre- and post-conditions of the operations.

Method fragments are defined with the class schema of Object-Z. For defining product structures, a *method fragment* has *attributes* as state variables and *constraints* as logical formulas. Similarly, *operation*, defined by an operation schema of Object-Z for specifying manipulations on the products, has *variables* as input and output parameters and *predicates* as logical formulas which define the effect of the operations. Pre-conditions and post-conditions of the operation belong to predicate. Note that a method fragment has two relationships *inheritance* and *reference* with other method fragments. In this respect, Object-Z is remarkably different from other, non-object-oriented, Method Engineering languages.

The example in figure 2 depicts an Object-Z specification of a simple variant of Entity Relationship Diagramming. The class schema ERD_in_ConceptualLevel starts with the declaration of the conceptual structure of Entity Relationship Diagram. It contains a number of state variables, whose values express an instance of Entity Relationship Diagram class. They address the definition of the concepts of ERD, i.e. Entity, Relationship and Attribute. A state variable is defined by its name and its type, for instance a powerset of the abstract basic type entity. The associations between them can be defined by maps, in the example of the association between entities and attributes a finite map from attribute to entity (EntityAttribute). The state variable declarations, i.e. attribute declarations are separated with a horizontal line from the state invariants, which denote constraints with respect to the state variables.

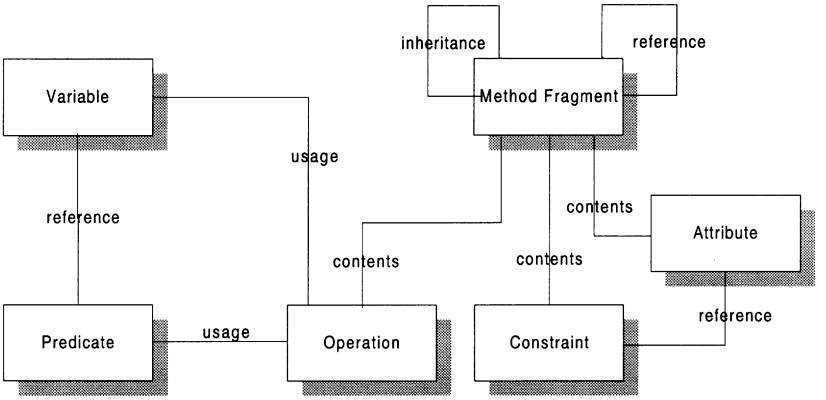

Figure 1 Meta-model of Object-Z.

For each permitted operation on the ERD_in_ConceptualLevel, such as IdentifyEntities, an operation schema is defined. Such a schema consists of variable declarations called signatures and predicates. The latter denote pre- and post-conditions for the operation. The Δ symbol indicates from which variables the values are changed by the operation. The ? symbol depicts input variables, and the variables with the prime represent the state after the operation.
The operations IdentifyEntities and IdentifyRelationships are for constructing conceptual ERDs and updating the state variables. For example, IdentifyEntities needs an input as a newly identified entity and adds it to the state variable Entity after its execution. This operation corresponds to the activity of identifying entities. The identified entities are stored in the state variable Entity. The operation IdentifyRelationships has a newly identified relationship and two entities participating in it as input parameters. The pre-condition, the first formula below the horizontal line of the operation schema, specifies that the two entities should be already identified before its execution. Therefore, this formula specifies implicitly the execution order of IdentifyEntities and IdetifyRelationships, IdentifyEntities should be executed before IdentifyRelationships. EntityRelationship_1 and EntityRelationship_2 stand for the association between identified entities and relationships.

The class schema ERD_in_ConceptualLevel does not include sufficient information to specify Entity Relationship Diagrams. For example, the schema did not have the information that an entity can be represented with a rectangle. The class schema ERD_in_TechnicalLevel , depicted in figure 3, specifies this *notational* information of ERD. It inherits state variables and operations from ERD_in_ConceptualLevel. The inheritance mechanism allows us to specify various level descriptions of method fragments separately. We define new state variables which include the information of graphical components standing for the ERD concepts. For example, entities in this level are defined by a map from conceptual entities to rectangles. Each Entity is represented with a rectangle. A graphical component such as a rectangle consists of x co-ordinate, y co-ordinate and a label, i.e. location information and its identifier. The operations, e.g. CreateEntity and MoveEntity are editorial manupulations of these graphical components. For example, MoveEntity moves the entity e? to x-axis-direction dx and y-axis direction dy.

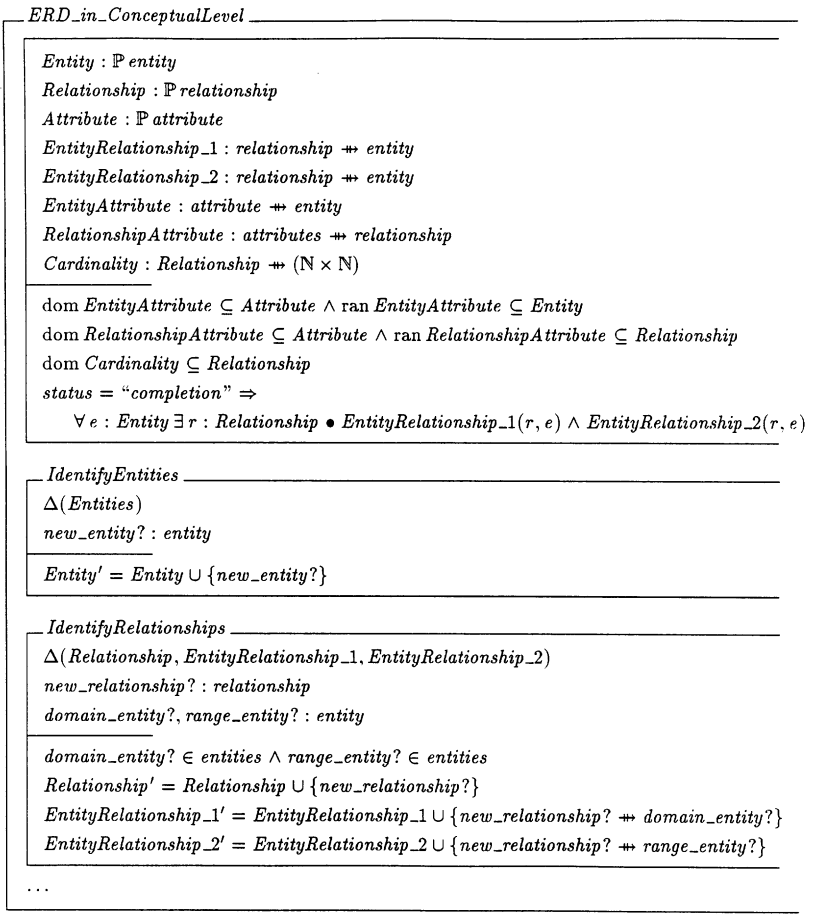

Figure 2 Description of conceptual ERD.

ERD_in_TechnicalLevel

ERD_in_ConceptualLevel

$entity\_symbols : Entity \nrightarrow Rectangle$
$attribute\_symbols : Attribute \nrightarrow Circle$
$relatioship\_symbols : Relationship \nrightarrow Diamond$
...

$\mathrm{dom}\ EntityGraphicalObject = Entity$
$\forall e : Entity \forall a : Attribute \bullet EntityAttribute(a) = e \Rightarrow$
$\quad connected(entity\_symbols(e), attribute\_symbols(a))$
$\forall r : Relationship \forall a : Attribute \bullet RelationshipAttribute(a) = r \Rightarrow$
$\quad connected(relationship\_symbols(e), attribute\_symbols(a))$
$\forall e1, e2 : Entity \forall rel : Relationship \bullet$
$\quad EntityRelationship\_1(rel) = e1 \wedge EntityRelationship\_2(rel) = e2 \Rightarrow$
$\quad\quad connected(relationship\_symbols(rel), entity\_symbols(e1)) \wedge$
$\quad\quad connected(relationship\_symbols(rel), entity\_symbols(e2))$

CreateEntity

$\Delta(entity\_symbols)x?, y? : Coordinate$
$label? : String$

IdentifyEntities
$entity\_graphical\_oject' = entity\_symbols \cup \{new\_object? \mapsto < x?, y?, label? >\}$

MoveEntity

$\Delta(entity\_symbols)$
$e? : Entity$
$dx?, dy? : Coordinate$

$\forall x, y : Coordinate, \forall label : String \bullet entity\_symbols(e?) =< x, y, label > \Rightarrow$
$\quad entity\_symbols' = entity\_symbols \cup \{e? \mapsto < x + dx?, y + dy?, label >\}$
...

...

$Rectangle ::= Coordinate \times Coordinate \times String$
...

Figure 3 Description of technical ERD.

3.2 MEL

MEL (Harmsen and Brinkkemper, 1995ab) is a language to describe and manipulate parts of IS development methods, and designed to support Method Engineering. The language offers representation mechanisms to describe methods on different levels of granularity. MEL facilitates both representation of method processes and method product models, as well as the tools that accompany a method. It is founded upon a definition in first order predicate logic. The product models, called product fragments, can be related to the process representations

(process fragments) through relationships, represented by MEL keywords. Such relationships include: *prerequisite*, which relates the product fragments required by a process fragment, and *manipulation*, relating the product fragments manipulated by a process fragment in a certain *role*, such as update, production, or usage. Product fragments can be involved in an *association*, or can *precede* each other. A product fragment can be associated to one or more *rules*, specifying a static constraint, and can be, on a higher granularity level, represented by a graphical symbol. Product fragments can be described in terms of *ontology* concepts and associations which is especially useful when assembling product fragments from different methods.

For process fragments, *decision* and *iteration* constructs are provided. Moreover, it is possible to specify parallelism. Process and product fragments are characterised by a number of predefined *property types* and *values*, the latter being taken from a number of pre-defined methodology-related value *domains*. Process and product fragments can be specialised, and it is possible to specify *abstract*, polymorphic product fragments (cf. templates) , which can be parametrised in the specialisation. Such polymorphic method fragments are also used to define role-specific *views*. Such a view is a specialisation which only contains the relevant parts for a specific actor type. Tool aspects are handled by providing special constructs, such as **SYMBOL**, and property types, such as colour and coordinates.

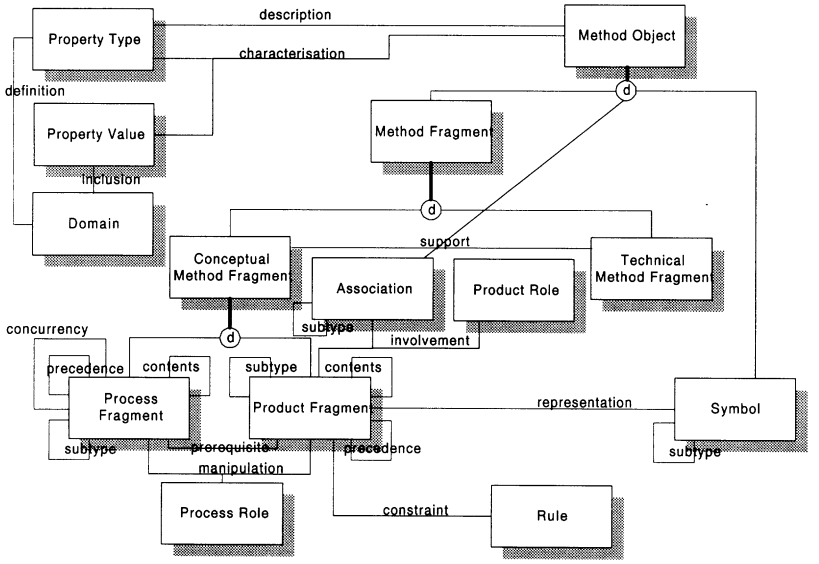

Figure 4 Meta model of MEL.

Figure 4 depicts a simplified meta model of the representational aspects of MEL. To improve readability, properties of the concepts, and cardinalities and role names of the relationship types are not shown. A *method object* can be a method fragment, an association, or a symbol. There are two relationship types between method object and property type: the ternary *characterisation* relationship, which is used to denote intrinsic properties of the method object

along with their values, and the binary *description* relationship, used to indicate properties of method object instances.

```
PRODUCT Simple EntityRelationshipDiagram:
 IS_A Product;
 LAYER Diagram;
 CREATOR TYPE Information Analyst;
 NAME TEXT;
 CREATION DATE DATE;
 DEFINITION TEXT;
 MANIPULATED BY {(Create ERD, Production), (Modify ERD, Update), (Create DFD, Usage)};
 (
  - Entity;
  - Attribute;
  - Relationship
    RULE r1: forall d in EntityRelationshipDiagram forall e in Entity exists r in Relationship [status(d) = 'completion' implies
    domain_of(e,r) or range_of(e,r)]
  # all entities are connected if ERD is completed #
 ).

PRODUCT Entity:
 LAYER Concept;
 SYMBOL Rectangle;
 MANIPULATED BY {(Identify provisional entities, Production), (Identify entities, Update), (Identify relationships, Usage)};
 ASSOCIATED WITH {(EntityRelationship_1, domain_of),(EntityRelationship_2, range_of), (EntityAttribute, has)}.

PRODUCT Attribute:
 LAYER Concept;
 SYMBOL Circle;
 ASSOCIATED WITH {(EntityAttribute, is_of),(RelationshipAttribute, is_of)}.

PRODUCT Relationship:
 LAYER Concept;
 SYMBOL Diamond;
 ASSOCIATED WITH {(EntityRelationship_1, has_domain), (EntityRelationship_2, has_range), (RelationshipAttribute, has)}.

ASSOCIATION EntityRelationship_1:
  ASSOCIATES(Entity, Relationship);
  CARDINALITY(1,n; 1,1) # the example requires only binary relationships #.

ASSOCIATION EntityRelationship_2:
  ASSOCIATES(Entity, Relationship);
  CARDINALITY(1,n; 1,1).

ASSOCIATION EntityAttribute:
  ASSOCIATES(Entity, Attribute);
  CARDINALITY(1,n; 0,n).

ASSOCIATION RelationshipAttribute:
  ASSOCIATES(Relationship, Attribute);
  CARDINALITY(1,n; 0,n).
```

Figure 5 Product representation in MEL.

Besides a representational part, which is the focus of this paper, MEL contains *operations* to administrate method fragments in the method base, to query them, and to assemble method fragments into a situational method. The examples below depict a product fragment and a process fragment. The simple ERD inherits all properties from a product fragment called "Product". Furthermore, it has the characterising properties layer and creator type. To enable further specification of its instances, it also has a number of properties which are not known until method application, such as name and creation date. It has some relationships with other method fragments, and it consists of three concepts, which are, together with their associations, specified further on.

The activities to produce the ERD are described in a process fragment specification. In process fragments, hyphens before activity names indicate sequential activities. By means of the **REQUIRED** and **DELIVERABLES** sections the input and output products, respectively, are indicated. Constructs not shown in the examples include parallelism, **DECISION**, to denote optionality and decisions, **REPEAT...UNTIL**, to denote iteration, and **AT LEAST ONE OF**, to denote non-determinism with respect to the required products.

```
PROCESS Create ERD:
 LAYER Diagram;
 REQUIRED Interview results
 (
  - Identify Entities;
  - Identify Relationships
 )
 DELIVERABLES Simple EntityRelationshipDiagram.
```

Figure 6 Process representation in MEL.

3.3 GOPRR

GOPRR (Kelly et al., 1996) focuses on the modelling of the conceptual structure of techniques and relationships between these, supporting:

- Decomposition and complex objects,
- Generalisation and specialisation of modelling concepts,
- Polymorphic modelling concepts,
- Representation independence, and
- Rules for checking the model integrity.

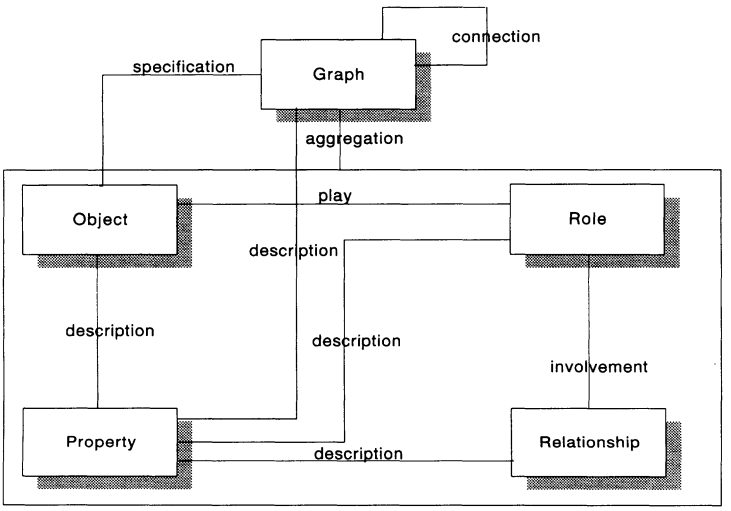

Figure 6 Meta model of GOPRR.

GOPRR-p (Koskinen, 1996) is used to model *process* structure of methods. It is an extension of GOPRR that conceptualises also the behavioural aspects of processes, in particular for automating the process model enaction. GOPRR-p is not taken into consideration in this comparison.

GOPRR distinguishes between objects, properties, relationships, roles, and graphs An *object* is used to model high granularity method products or concepts. A *relationship* is a connection between a set of objects. A *role* specifies how an object is connected to a relationship. A set of elements is collected using a *graph*, which enables the coupling of diagrams to diagrams, as well as concepts to diagrams. The relationships between property and the other concepts all relate to the instance level, and are therefore similarly named as in the **MEL** meta-model.

In the example below, boxes represent objects. Circles represent roles, diamonds represent relationships, and ellipses represent properties describing instances of objects. Properties characterising an object (method fragment) are placed within a dotted rectangle. For instance, the object Entity has three roles (domain_of, range_of, and has), which are associated with relationships. The entire model, a graph, has three descriptive properties: Name, Creation Data and Definition, which are used to characterise an instance of this graph. It has two characterising properties: Layer, and Creator Type. These properties characterise the graph itself. Something which is not shown in the example is, that properties can be shared by several objects and relationships.

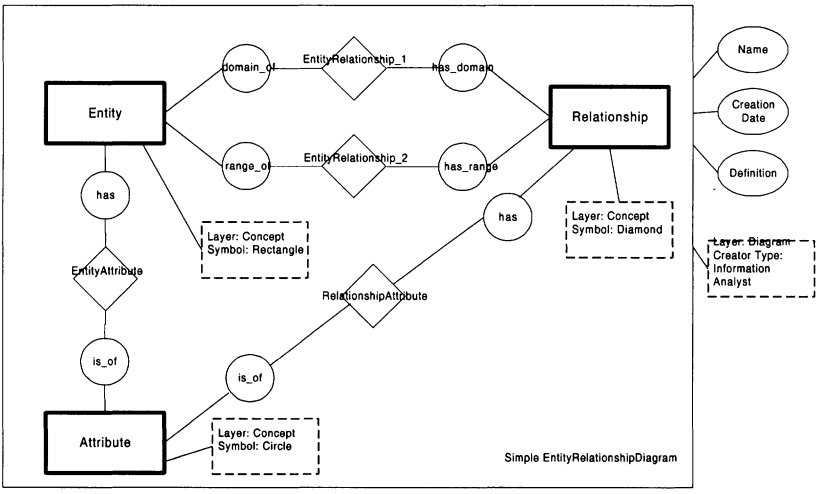

Figure 7 Product representation in GOPRR.

3.4 HFSP

HFSP (Katayama, 1989; Song and Osterweil, 1992) provides a language for describing software process programs in a process-centred style. It is based on attribute grammars. The descriptions, i.e. HFSP programs, consist of two parts - declarations of data types and the definition of derivation rules that specify the activities and the computation of products in the

processes. Data types are defined in a similar way as in well-known program languages, such as record type, set type, enumeration type and so on.

The meta model of HFSP is shown in Figure 8. A *type* is a definition of a product type and specifies the type of *attribute* values associated with *activity*. An activity corresponds to a non-terminal symbol or a terminal symbol in grammatical *derivation rules*. Typical derivation rules have the following form:

$A \Rightarrow B_1\ B_2 \ldots B_n$
when *conditions*
where *computation rules*

The activity A is decomposed to sub activities B_1, B_2, ... and B_n. A is a non-terminal symbol and each B_i ($1 \leq i \leq n$) corresponds to non-terminal or terminal symbols. A and each B_i can have inputs and outputs as their attributes. When A has several inputs and outputs, we write $A(in_1, in_2, ..., in_m \mid out_1, out_2, ..., out_k)$. Intuitively speaking, in_1,..., in_m are inherited atributes, while out_1, ..., out_k are synthesised attributes, because they are computed from the values associated with B_1, ..., B_n and in_1, ... , in_m.

We use the term *decomposition rule* to denote the right hand side of derivation rules that have on the left hand side the activities in which an activity can be decomposed. Thus, a decomposition rule consists of a sequence of *activities*, and the *usage* relationship expresses which activities occur in the decomposition rule. *Computation rule* and *condition* are associated with a derivation rule. The former is for computing an output attribute value associated with the activity when it is decomposed. The latter specifies the condition which should hold when the activity is decomposed, i.e. executed.

In the example depicted in figure 9, which is part of an ER diagram description in HFSP, we define the structure of ER diagram by using set type and record type constructions. For example, EntityRelationship_1 (association between Entity and Relationship) is defined as a pair (record type) of entities and relationships. Conceptual ER diagrams consist of a set of entities, a set of relationships, a set of attributes and their associations (named EntityRelationship_1, EntityRelationship_2, EntityAttribute and RelationshipAttribute). To attach graphical information to entity, relationship and attribute components on diagram notation, we introduce the other types entity_symbols, relationship_symbols and attribute_symbols using record type construction. They have graphical components such as a rectangle, a diamond and a circle.

The activities for constructing ER diagrams are specified by derivation rules associated with conditions and computation rules. As mentioned before, conditions specify the constraints for application of the derivation rule. The input and output values are computed following the computation rules. In the example of the second rule of IdentifyEntity, only if the value of Entity.e is not included in entities.in, the rule is applied and then the value of entities.out is computed as the union of entities.in and {Entity.e}. Note that we use the variables for denoting the attribute values, i.e. the variables keep the referential transparency. Because of this property, we often write many copy rules to propagate the attribute values that are globally used. A variable name may be prefixed with its type. For example, entities.in stands for the variable "in" whose type is entities.

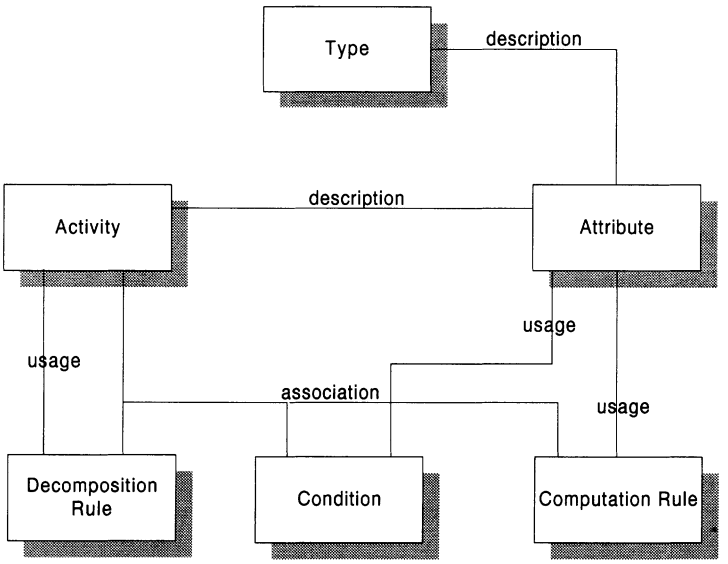

Figure 8 Meta model of HFSP.

As the execution of activities proceeds, the derivation tree becomes larger. The derivation tree records the execution history. The execution order of activities is the derivation order of the rules and the derivation order is specified by the conditions and the computation rules. When the conditions are satisfied and where the values necessary for the computation are already determined, we can execute the derivation, i.e. the corresponding activity. Thus it means that HFSP specifies the execution order of activities implicitly. Furthermore HFSP can specify concurrency and non-determinism on activity execution. Suppose there is more than one node the derivation rules can be applied in the derivation tree. They can be derived concurrently, i.e. parallel parsing, or one or some of them can be selected and derived in a non-deterministic way.

As shown in the example, method-descriptions cannot be separated among different levels. Therefore, notational information of ERD is scattered in its conceptual description.

Note that it is difficult in HFSP to specify constraints on products because its data type definition part has no syntactic devices to specify them. If a constraint has to be specified on the products, it has to be associated with derivation rules. This is also one of the reasons why the method descriptions in HFSP are not so comprehensive.

```
type
  entities : set of Entity
  relationships : set of Relationship
  attributes : set of Attribute
  EntityRelationship_1, EntityRelationship_2 : (Entity, Relationship)
  EntityAttribute : (Entity, Attribute)
  RelationshipAttribute : (Relationship, Attribute)
  entity_symbols : set of (Entity, Rectangle)
  relationship_symbols : set of (Relationship, Diamond)
  attribute_symbols : set of (Attribute, Circle)
  Rectangle, Diamond, Circle : (Coordinate, Coordinate, Label)
  ERDiagram : (entity_symbols, relationship_symbols, attribute_symbols,
          EntityRelationship_1, EntityRelationship_2,
          EntityAttribute, RelationshipAttribute)

activity
  MakeERDiagram( | ERDiagram.out) ⇒
    DrawEntity( | entity_symbols)
    DrawRelationship(entity_symbols.out | relationship_symbols.out,
                EntityRelationship_1.out, EntityRelationship_2.out)
    DrawAttribute(entity_symbols.out, relationship_symbols.out |
              attribute_symbols.out, EntityAttribute.out,
                Relationship_attribute.out)
    where
     ERDiagram.out = (entity_symbols.out, relationship_symbols.out,
                attribute_symbols.out, EntityRelationship_1.out,
                EntityRelationship_2.out, EntityAttribute.out,
                RelationshipAttribute.out)

  DrawEntity( | entity_symbols.out) ⇒
    IdentifyEntity( | entities.out)
    CreateEntity(entities.out | entity_symbols.in)
    MoveEntity(entity_symbols.in | entity_symbols.out)

  IdentifyEntity( | entities.out) ⇒ ε
    where entities.out = { }

  IdentifyEntity( | entities.out) ⇒ IdentifyEntity( | entities.in)
    when  Entity.e ∉ entities.in
    where entities.out = entities.in ∪ {Entity.e}

  CreateEntity(entities.in | entity_symbol.out) ⇒ ε
    when  entities.in = { }
    where entity_symbols.out = { }

  CreateEntity(entities.in | entity_symbols.out) ⇒ CreateEntity(entities.out | entity_symbols.in)
    when  entities ≠ { } ∧ entity.e ∈ entities.in
    where entities.out = entities.in - {Entity.e}
       entity_symbols.out = entity_symbols.in ∪ {(Entity.e, (x,y,label)}

  MoveEntity(entity_symbols.in | entity_symbols.out) ⇒ ε
    where entity_symbols.out = entity_symbols.in

  MoveEntity(entity_symbols.in | entity_symbols.out) ⇒ MoveEntity(entity_symbols.in | entity_symbols.mid)
    when  (Entity.e, (x,y,label)) ∈ entity_symbols.mid
    where entity_symbols.out = entity_symbols.mid - {(Entity.e, (x,y,label))}
                    ∪ {(Entity.e, (x+dx,y+dy,label))}
```

Figure 9 Method fragment representation in HFSP.

4 COMPARING THE METHOD ENGINEERING LANGUAGES

In table 1 the four languages are characterised on the basis of the following aspects:

- *Basis*, which is the underlying -mathematical- formalism or modelling language.
- *Scope*, which denotes the range of applications for which the language is being used.
- *Paradigm*, addressing the philosophy or "way of thinking" a Method Engineering language adopts.
- *Explicitness*, which indicates whether method fragments are completely described, or that part of the method fragment specification should be derived. For instance, the activity sequence can be derived from pre- and post-conditions associated with process fragments.
- *Focus*, indicating which part or aspect of a method specification is particularly emphasised by the language.
- *Size*, which gives an indication of the average size of a method specification in a language.

Table 1 Characterisation of Method Engineering languages

	Object-Z	*MEL*	*GOPRR*	*HFSP*
basis	ZF set theory	predicate logic	extended ER model	attribute grammars
scope	general-purpose	Method Engineering	meta-modelling	process modelling
paradigm	object-oriented	data/process-oriented	data-oriented	functional
explicitness	process order implicit	explicit	explicit	implicit
focus	method products	method products	method products	method processes
size	moderate	large	moderate	large

Table 2 represents an comparison of the four languages by assessing the extent to which each language meets the requirements stated in section two.

Object-Z does support process representation, but, due to its object-oriented roots, only in combination with a product description. HFSP only supports record-like product descriptions, and is therefore less suitable for representing products. All languages can support both the development and the project management perspective, but none of them offers special constructs to distinguish or relate the two. Only MEL offers support for representation of technical method fragments by providing the **SYMBOL** object class and several property types for technical method fragments. However, all other language are capable of representing them,

as has been shown in some examples. Only GOPRR is not able to express formal rules; HFSP offers conditions, MEL rules, and Object-Z formulas. Modularisation is a weak point of HFSP; all other languages have constructs to support this feature: Object-Z by class schema's and their instances, MEL by method objects, and GOPRR by graphs. The investigated Method

Table 2 Comparison of Method Engineering languages

	Object-Z	MEL	GOPRR	HFSP
suitability	no methodology-specific concepts, relatively hard to learn	many methodology-specific concepts, relatively hard to learn	concepts for modelling techniques, easy to learn	non-determinism, simulation, hard to learn
processes & products	both, but processes less well supported	both	products	both, but products less well supported
development & proj. mgmt.	mainly development	mainly development	development	mainly development
conceptual & technical	both	both, special constructs and property types	both	both
formal rules	yes	yes	no	yes
actors	no, but can be user-defined	as pre-defined property types	no	no
type/instance	yes	no	no	no
non-determinism	yes	no	no	yes
parallellism	yes	yes	no	yes
design rationale	no, but can be user-defined	no	no	no
modularisation	yes (objects, classes)	yes	yes (composite concepts)	no
views	no	yes	no	no
unambiguity	unambiguous	unambiguous	unambiguous	unambiguous

Engineering languages do not provide explicit support for actors or roles. MEL features pre-defined property types such as "creator" or "responsible" which serve as actor and role representations. In Object-Z, actors and roles can be represented by object classes. Only in Object-Z, the difference between types and instances is handled well. GOPRR and HFSP can represent both types and instances, but not their relationships. MEL is not able at all to represent method fragment instances. Non-determinism of operation sequence is supported in Object-Z and HFSP. GOPRR does not operations at all, in MEL the sequence of operations is explicitly defined. Parallelism is supported by all three languages that provide representation

for process fragments. None of the investigated languages provides support for capturing design rationale, although in Object-Z object class schemata to serve that purpose can be defined. MEL supports the explicit definition of views by specialisation of generic method fragments, the other languages do not. All of the languages are unambiguously and formally defined by their authors, leaving no room for misinterpretations. Object-Z provides no methodology-specific concepts, which causes the language somewhat harder to learn for methodologists. Due to the compact representation, the size of Object-Z specifications does not suffer from this fact. MEL provides a lot of methodology-specific concepts and properties. This huge number makes the language also harder to learn. GOPRR only addresses product representation (GOPRR-p was not yet taken into consideration), but does this in an easy and elegant fashion. GOPRR is in particular suited for modelling modelling techniques, and not complete methods, although the graph concept enables to do so. HFSP is particularly suitable for modelling non-deterministic processes. Due to the precise description of processes, HFSP is executable and can be used for simulation purposes. This language is relatively difficult to learn.

5 CONCLUSION

In this paper we proposed a number of requirements for Method Engineering languages. From each of the four identified categories of Method Engineering languages we have taken one example, which have been described with meta-models and illustrated with example specifications. We have compared the four languages on the basis of the requirements.

A general conclusion that can be drawn is that there is no ultimate Method Engineering language. Choice of the language depends on the desired purpose and goals one wants to reach. Also in this respect, Method Engineering languages and their usage are very similar to IS modelling techniques. Object-Z provides compact, elegant specifications underpinned by the object-oriented paradigm. If only the process aspects of a method need to be represented, Object-Z is less suitable. MEL tries to combine the product and process aspects and uses methodology-specific terms, particularly in its pre-defined properties. Because MEL was designed by looking at advantages of other Method Engineering languages, it is probably the best all-round language around. However, the huge number of concepts and properties make the language hard to learn, which counteracts the advantage of being methodology-specific for a part. GOPRR is not so hard to learn, but addresses only the product aspect. This language is best suited for modelling and representing modelling techniques and their interrelationships. HFSP is particularly suited for modelling processes, the product representation support by record types is quite rudimentary. HFSP is at its best if processes need to be simulated, for instance to calculate various alternative project plans.

The conclusion that there is no ultimate Method Engineering language could lead to a situation in which Method Engineering languages are composed of fragments originating from several Method Engineering languages, to obtain a purpose-fit language. We call this *Method Engineering of Method Engineering languages.*

Further research focuses on refining the various categories and comparing more languages. Purposes of Method Engineering will be related to the various languages around. The comparison techniques will be sophisticated and formalised. Currently a reference framework and associated comparison metrics for Method Engineering languages is under development. This framework is developed towards a Method Engineering ontology, i.e. a formal data model

containing basic concepts of IS engineering methods, while taking into account earlier results (described in (Olle et al., 1991) and (Heym and Österle, 1992)) regarding such data models.

6 REFERENCES

Benali, K., Boudjlida, N., Charoy, F., Derniame, J.-C., Godart, C., Griffiths, Ph., Gruhn, V., Jamart, Ph., Oldfield, D. and Oquendo, F. (1989) Presentation of the ALF project. *Proceedings of the International Conference on System Development Environments and Factories*, Berlin.

Brinkkemper, S. (1991) Formalisation of Information Systems Modelling, Dissertation University of Nijmegen, Thesis Publishers, Amsterdam.

Duke, R., King, P., Rose, R. and Smith, G. (1991) The Object-Z Specification Language, Technical Report 91-1, Software Verification Centre, University of Queensland.

Emmerich, W., Junkermann, G. and Schäfer, W. (1991) MERLIN: knowledge based process modelling. *First European Workshop on Software Process Modelling*, Milan.

Harmsen, F., Brinkkemper S. and Oei H. (1994) Situational Method Engineering for Information System Projects. *Proceedings of the IFIP WG8.1 Working Conference CRIS'94* (Eds. T.W. Olle and A.A. Verrijn-Stuart), North-Holland Publishers, Amsterdam, pp. 169-194.

Harmsen, F., and Brinkkemper, S. (1995a) Description and Manipulation of Method Fragments for Situational Method Assembly. *Proceedings of the Workshop on Management of Software Projects*, Pergamon Press, London.

Harmsen, F. and Brinkkemper S. (1995b) Design and Implementation of a Method Base Management System for a Situational CASE Environment. *Proceedings of the 2nd Asian-Pacific Software Engineering Conference (APSEC'95)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 430-438.

Heym, M. and Österle, H. (1992) A reference model of information systems development. *The Impact of Computer Supported Technologies on Information Systems Development* (Eds. K.E. Kendall, K. Lyytinen and J.I. DeGross), Amsterdam, North-Holland, pp. 215-240.

Hofstede, A.H.M. ter (1993), Information modelling in data intensive domains, dissertation University of Nijmegen, the Netherlands.

Hong, S., van den Goor, G., and Brinkkemper, S. (1993), A Comparison of Object-Oriented Analysis and Design Methodologies. *Proceedings of the 26th Hawaiian Conference on System Sciences (HICSS-26)*, IEEE Computer Science Press.

Iivari, J. (1994) Object-oriented information systems analysis: A comparison of six object-oriented analysis methods. *Proceedings of the IFIP WG8.1 Working Conference CRIS'94* (Eds. T.W. Olle and A.A. Verrijn-Stuart), North-Holland Publishers, Amsterdam, pp. 85-110.

Katayama, T. (1989) A hierarchical and functional software process description and its enaction. *Proceedings of the 11th Int. Conf. on Software Engineering*. pp.-343-352.

Kelly S., Lyytinen, K. and Rossi, M. (1996) MetaEdit+ A Fully Configurable Multi-User and multi-Tool CASE and CAME Environment. *Proceedings of the CAiSE'96 conference*, 20-24 May, Heraklion, Crete, Greece.

Koskinen, M. (1996) Designing Multiple Process Modelling Languages for Flexible, Enactable Process Models in a MetaCASE Environment, *Proceedings of the 7th European Workshop on Next Generation CASE Tools (NGCT'96)*, Heraklion, Crete, Greece.

Kumar, K. and Welke, R.J. (1992) Methodology Engineering: A proposal for Situation-specific Methodology Engineering. *Challenges and Strategies for Research in Systems Development* (Eds. W.W. Cotterman and J.A Senn), John Wiley and Sons Ltd., pp. 257-269.

Martin, J. (1990) Information Engineering, Book II - Planning and Analysis, Prentice-Hall, Englewood Cliffs.

Marttiin, P., Rossi, M., Tahvanainen, V.-P. and Lyytinen, K. (1993) A Comparative Review of CASE Shells: a preliminary framework and research outcomes. *Information and Management*, **25**, pp. 11-31.

Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M. (1990) Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, **8**, 4, pp. 325-362.

Oei, J.L.H. and E.D. Falkenberg (1994) Harmonisation of Information System Modelling and Specification Techniques. *Proceedings of the IFIP WG8.1 Working Conference CRIS'94* (Eds. T.W. Olle and A.A. Verrijn-Stuart), North-Holland Publishers, Amsterdam, pp. 151-168.

Olle, T.W., Sol, H.G. and Tully, C.J. (Eds.) (1983) *Information Systems Design Methodologies: A Feature Analysis*. Elsevier Science Publishers, North-Holland, Amsterdam.

Olle, T.W., Hagelstein, J., MacDonald, I.G., Rolland, C., Sol, H.G., Van Assche, F.J.M. and Verrijn-Stuart, A.A (1991) *Information Systems Methodologies: A framework for understanding*. Addison-Wesley Publishing Company, Wokingham, England.

Saeki, M., Kaneko, T., and Sakamoto, M. (1991) A Method for Software Process Modeling and Description using LOTOS. *Proceedings of the 1st International Conference on the Software Process* (Ed. M. Dowson), IEEE Computer Society Press, Los Alamitos, CA, pp. 90-104.

Saeki, M., and Wen-yin, K. (1994) Specifying Software Specification and Design Methods. *Advanced Information Systems Engineering* (Eds. G. Wijers, S. Brinkkemper and T. Wasserman), LNCS#811, Springer-Verlag, pp. 353-366.

Smolander, K. (1992) OPRR - A Model for Methodology Modeling. *Next Generation of CASE Tools* (Eds. K. Lyytinen and V.-P. Tahvanainen), Studies in Computer and Communication Systems, IOS press, pp. 224-239.

Slooten, K. van, and Brinkkemper S. (1993) A Method Engineering Approach to Information Systems Development. *Proceedings of the IFIP WG8.1 Conference on Information Systems Development Process* (Eds. N. Prakash, C. Rolland and P. Pernici), Como, pp. 167-186.

Song, X., and Osterweil, L.J. (1992), Towards objective, systematic design-method comparison. *IEEE Software*, **34**, 5, May, pp. 43-53.

Sorenson, P.G., Tremblay, J-P. and McAllister, A.J. (1988) The Metaview system for many specification environments. *IEEE Software*, **30**, 3, March, pp. 30-38.

Sowa, J.F., and Zachman, J.A. (1992) Extending and formalizing the framework for information systems architecture. *IBM Systems Journal*, **31**, 3, pp. 590-616.

Venable, J. (1993) CoCoA: A Conceptual Data Modelling Approach for Complex Problem Domains. Ph.D. dissertation, State University of New York, Binghampton.

Verhoef, T.F. and Ter Hofstede, A.H.M. (1995) Feasibility of Flexible Information Modelling Support. *Advanced Information Systems Engineering* (Eds. J. Iivari, K. Lyytinen and M. Rossi), LNCS #932, Springer-Verlag, pp. 168-185.

Wijers, G. and Dort, H. van (1990) Experiences with the use of CASE tools in the Netherlands. *Advanced Information Systems Engineering* (Eds. B. Steinholz, A. Sølvberg and L. Bergman), LNCS#436, Springer-Verlag, pp. 5-20.

Wijers, G. (1991) Modelling Support in Information Systems Development. Ph.D. dissertation, Thesis publishers, Amsterdam.

7 BIOGRAPHY

Frank Harmsen is a researcher in the Information Systems Design Methodology Research Group at the Computer Science Department of the University of Twente in the Netherlands. He holds a B.Sc and M.Sc in Mathematics and Computer Science from the University of Nijmegen. His research interests are information system methodology, meta-modelling, Method Engineering, and CASE tools, about which he has published several papers. Current research activities focus on defining formalisms and tools for representation and assembly of method fragments for Situational Method Engineering. He was co-editor of the 1993 edition of the Workshop on Next Generation of CASE Tools (NGCT), and served on the organisation committee of CAiSE'94 (Conference on Advanced Information Systems Engineering). He is a member of the Netherlands Society for Informatics.

Motoshi Saeki is an associate professor of Tokyo Institute of Technology, Tokyo, Japan. He received a Ph.D degree from Dept. of Computer Science, Tokyo Institute of Technology in 1983. He has worked for Dept. of Computer Science as a research associate and since 1988 as an associate professor. His current interests include specification & design methods, formal methods (in particular, application of formal methods), human factor in software development and CSCW in software development.