Pergamon

Information Systems Vol. 24, No. 3, pp. 209–228, 1999

Printed in Great Britain
0306-4379/99 $20.00 + 0.00

PII: S0306-4379(99)00016-2

META-MODELLING BASED ASSEMBLY TECHNIQUES FOR SITUATIONAL METHOD ENGINEERING†

SJAAK BRINKKEMPER[1], MOTOSHI SAEKI[2], AND FRANK HARMSEN[3]

[1]Baan Company R & D, P.O. Box 143, 3770 AC Barneveld, The Netherlands
[2]Tokyo Institute of Technology, Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan
[3]Ernst & Young Consulting, P.O. Box 3101, 3502 GC Utrecht, The Netherlands

(Received 12 October 1998; in final revised form 10 April 1999)

Abstract — Method engineering for information system development is the discipline to construct new advanced development methods from parts of existing methods, called method fragments. To achieve this objective, we need to clarify how to model the existing methods and how to assemble method fragments into new project-specific methods, so-called situational methods. Especially, to produce meaningful methods, we should impose some constraints or rules on method assembly processes. In this paper, we propose a framework for hierarchical method modelling (meta-modelling) from three orthogonal dimensions: perspectives, abstraction and granularity. According to each dimension, methods and/or method fragments are hierarchically modelled and classified. Furthermore, we present a method assembly mechanism and its formalization as a set of rules. These rules are both syntactic and semantic constraints and presented in first order predicate logic so that they can play an important role in the assembly process of syntactically and semantically meaningful methods from existing method fragments. The benefit of our technique is illustrated by an example of method assembly, namely the integration of the Object Model and Harel's Statechart into Objectcharts.

Key words: Method Engineering, Meta-Modelling, Method Assembly

1. INTRODUCTION

The size and complexity of projects for developing information systems are becoming larger and more complicated. Therefore, development methods and supporting tools turn out to be one of the most significant key factors to achieve great success of development projects. Until now, many methods such as structured analysis/design [8] and object-oriented analysis/design [20] have been proposed and many textbooks have been published. The information-technology industry is putting the existing methods and corresponding supporting tools into practice in real development projects. However, much time and effort is spent on applying the methods effectively in these projects. One of the reasons is that contemporary methods are too general and include some parts, which do not fit to the characteristics of real projects and their contexts. In fact, according to [22] that investigated the activities to apply methods to real projects, many practitioners took time to devise the adaptation of methods to their projects before starting the projects. To enhance the effect of methods, for each of real projects, we need to adapt the methods or construct the new ones so that they can fit to the project.

Method Engineering, in particular Situational Method Engineering [4, 10, 13] is the discipline to build project-specific methods, called *situational methods*, from parts of the existing methods, called *method fragments*. This technique is coined *method assembly*. In fact, many methods can be considered to be the result of applying method assembly. For instance, OMT [20] has been built from the existing fragments Object Class Diagram (extended Entity Relationship Diagram), State Transition Diagram, Message Sequence Chart and Data Flow Diagram, all originating from other method sources. This example shows that method assembly could produce a powerful new method from the existing method fragments.

To assemble method fragments into a meaningful method, we need a procedure and representation to model method fragments and impose some constraints or rules on method assembly processes. If we allow assembly arbitrary method fragments, we may get a meaningless method. For example, it makes no sense to assemble Entity Relationship Diagram and Object Class Diagram in the same level of abstraction. Thus, the modelling technique for method fragments, so called meta-modelling technique should be able to include the formalization of this kind of constraints or rules to avoid producing

†Recommended by Barbara Pernici and Costantino Thanos

meaningless methods.

Several researchers applied adequate meta-modelling techniques based on Entity Relationship Model [3, 18, 26], Attribute Grammars [15, 25], Predicate Logic [3, 18, 19] and Quark Model [1] for various method engineering purposes (see Section 6). Some of these works discuss the inconsistency of products when we assemble several methods into one. However, none of them referred to the method assembly function itself yet. Song investigated existing methods, such as OMT and Ward/Mellor's Real Time SDM [30], and classified the way various methods are put together [25]. Guidelines or rules to assemble methods were not elaborated in this study. Furthermore, as discussed later in Section 6, his classification is fully included in ours.

In this paper, we propose a framework for hierarchical meta-modelling from three orthogonal dimensions: perspective, abstraction and granularity. According to each dimension, methods and method fragments are hierarchically modelled and classified. According to this classification of method fragments, we can provide the guideline for meaningful method assembly. That is to say, we can suggest that method fragments, which belong to a specific class, can be meaningfully assembled. For example, we can sufficiently construct a meaningful method from method fragments with the same granularity level. In another example, it is not preferable to assemble the method fragments belonging to the same specific category such as Entity Relationship Diagram and Object Class Diagram, as the latter can be seen as an extension of the former. These kinds of guideline and constraints can be formalized as a set of rules based on our multiple hierarchical dimensions. These rules can be presented in first order predicate logic and play an important role on clarifying method assembly mechanism.

This paper is organised as follows. In the next section, we begin with illustrating a simple example of the method fragment Statechart and introduce three orthogonal dimensions for classification of method fragments. We also discuss the semantics of method fragments that is provided by an ontology technique in the section. Section 3 presents method assembly by using an example of assembling Object Model and Statechart into the new method fragment Objectchart. This example suggests to us what kind of guidelines or constraints are required to method assembly. Based on the guidelines and/or constraints that we extracted from this example, we can list up the requirements of method assembly, and generalize these guidelines and constraints following the requirements and formalize them in Section 4. Sections 5 and 6 summarize related work and our work respectively.

2. A CLASSIFICATION FRAMEWORK FOR METHOD FRAGMENTS

2.1. Method Fragment Example in the Product Perspective

We begin with an example of the description of the method fragment of Harel's Statechart. Statecharts can be seen an extension of finite state transition diagram to specify reactive systems [9]. To avoid the explosion of the number of states occurring when we specify complicated systems with usual state transition machines, it adopted two types of structuring techniques for states, i.e. hierarchically decomposition of states: one is called AND decomposition for concurrency, and the other one is OR decomposition for state-clustering. The description of the method fragment is illustrated in the meta-model in Figure 1 in the notation of Entity Relationship Attribute Diagrams. (To avoid confusion, we use the terms concept, association and property in method fragments instead of entity, relationship and attribute.) Roughly speaking, any method has two fundamental elements; one is products and their structures, and the other one is procedures and their execution order to develop the products. On account of brevity, we pick up the only conceptual structure of Statechart products in this figure. The word "conceptual" means that the description does not include the notation of Statechart diagrams, e.g. we use ovals for expressing states, etc.

The Statechart technique comprises four concepts: State, Transition, Event and Firing condition. If a firing condition associated with a transition holds, the transition can occur and the system can change a state (called source state) to a destination state. During transition, the system can output or send an event to the other Statecharts. Firing conditions can be specified with predicates and/or receipt of these events. So we can have four associations among the three concepts, and two associations on the state concept for expressing AND decomposition and OR decomposition. Note that the meta-model does not include representational information, e.g. a state is represented in a rounded box in a diagram, and events are denoted by arrows. We define this kind of information as another aspect of method modelling and discuss it in the next section.

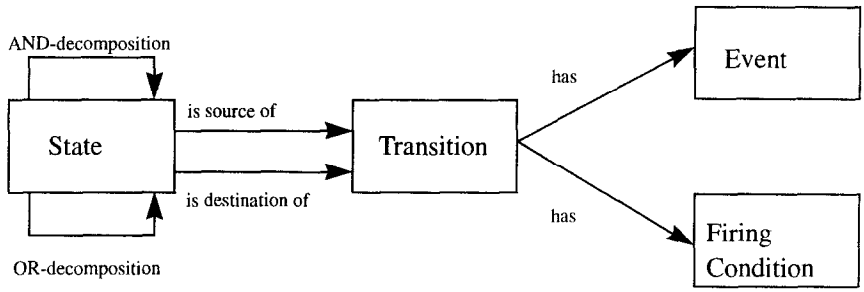

Fig.1: Statechart Method Fragment

2.2. *Classification of Method Fragments*

Method fragments are classified according to the dimensions *perspective*, *abstraction level,* and *layer of granularity.*

First, the perspective dimension of the classification considers the *product* perspective and the *process* perspective on methods. Product fragments represent deliverables, milestone documents, models, diagrams, etc. Process fragments represent the stages, activities and tasks to be carried out. As mentioned in Section 2.1, Figure1 is a description of Statechart method in the product perspective.

The abstraction dimension comprises the *conceptual level* and the *technical level.* Method fragments on the conceptual level are descriptions of information systems development methods or part thereof. Technical method fragments are implementable specifications of the operational parts of a method, i.e. the tools. Some conceptual fragments are to be supported by tools, and must therefore be accompanied by corresponding technical fragments. One conceptual method fragment can be related to several external and technical method fragments. The conceptual method fragment of Statechart in the product perspective has been shown as Figure 1, whereas the examples of the corresponding technical fragments are notation of Statechart diagrams and the STATEMATE tool for editing Statecharts [9].

One of the most important and main discriminating properties of method fragments is the *granularity layer* at which they reside. Such a layer can be compared with a decomposition level in a method. A method, from the process perspective, usually consists of stages, which are further partitioned into activities and individual steps. A similar decomposition can be made of product fragments, with the entire system at the top of the tree, which is subsequently decomposed into milestone deliverables, model, model components, and concepts. Research into several applications of method engineering [3, 5] shows that methods can be projected on this classification. A method fragment can reside on one of five possible granularity layers:

- **Method**, which addresses the complete method for developing the information system. For instance, the *Information Engineering* method resides on this granularity layer.
- **Stage**, which addresses a segment of the life-cycle of the information system. An example of a method fragment residing on the Stage layer is a *Technical Design Report.* Another example of a Stage method fragment is a CASE tool supporting Information Engineering's *Business Area Analysis* [17] stage.
- **Model**, which addresses a perspective [19] of the information system. Such a perspective is an aspect system of an abstraction level. Examples of method fragments residing on this layer are the *Data Model*, and the *User Interface Model.*
- **Diagram**, addressing the representation of a view of a Model layer method fragment. For instance, the *Object Diagram* and the *Class Hierarchy* both address the data perspective, but in another representation. The *Statechart* resides on this granularity layer, as well as the modelling procedure to produce it.
- **Concept**, which addresses the concepts and associations of the method fragments on the Diagram layer, as well as the manipulations defined on them. Concepts are subsystems of Diagram layer method fragments. Examples are: *Entity*, *Entity is involved in Relationship*, and *Identify entities*

2.3. Semantics of Method Fragments

Description of semantics of method fragments is one of the major problems in Situational Method Engineering. To alleviate semantic problems, method fragments are described in terms that are defined as complete and unambiguous as possible. Assembly should also be based on the semantics, and ideally pragmatics, of each method fragment involved, rather than on abstract or concrete syntax. One way to achieve this, is to characterize method fragments with as many properties as possible. The problem with this approach is, however, that there are few relationships defined between properties. Moreover, semantics of property value types are in most cases of a rather coarse granularity, which makes them less suitable to provide method fragment semantics. And third, completeness of a description in terms of individually defined properties is hard to be proved.

Some researchers proposed how to provide formal semantics for method fragments and some of them have used an ontology or an anchoring system [6]. In this approach, method fragments can be explained in a common terminology, where each term expresses an atomic semantic primitive and we have a common and unambiguous interpretation for it [13, 27, 29]. As an example, consider the meaning of the concept "Attribute" that appears in the Object Model in Figure 3. According to the well-known definition, it is a *property* that objects belonging to a specific class have. It also has a *set of data* whose element is set to it at a point of time, and the element, i.e. data value *describes* the object at that time. Thus we can have the terms "property", "data set" and "description (describe)" as semantic primitives and use them to express the meaning of "Attribute".

The way to extract and classify semantic primitives of methods, i.e. common terminology, is one of the important issues in this kind of techniques. Although what semantic primitives on the ontology we should use is out of scope in this paper, we should show how we can use this kind of system in our assembly technique. We adopted the MDM (Methodology Data Model) that has been defined by Harmsen [13], since it is suitable for diminishing the above issues. In the rest of this subsection, we introduce his system very briefly, however sufficiently to understand how to embed semantic aspects of method assembly to our framework.

Method fragments should be *anchored*, i.e. described in terms of unambiguously defined concepts and, possibly, associations of an anchoring system. An anchoring system of method fragments is a restricted set of well-defined atomic primitives. Figure 2 illustrates how to provide the semantics of method fragments by using an anchoring system. Method fragments consist of three sets; a set of Concepts CN, a set of Associations A and a set of Attributes At, as shown in the example of Figure1. Each of their elements is mapped into a set of the semantic elements on an anchoring system with the map α. For example, α("Attribute") = {property, data set, description} where "Attribute" is a concept of Object Model method fragment, and property, data set and description are semantic concepts on the anchoring system. The semantic concepts constitute a network with the function Φ and it restricts the meaning of associations among the concepts of method fragments, such as "has" in Figure1.

The anchoring system can be formally defined as follows: Let Γ be the anchoring system. α: $M \rightarrow \wp\Gamma$, the anchoring or interpretation function, maps method fragments in M on a subset of the anchoring system. Because mappings need to be unambiguous, α is a bijection. In principle, the anchoring system prescribes the set of possible method fragments, and is therefore limitative.

An ontology for method fragments is an anchoring system $\Delta = <CN_O, A_O, \Phi>$ where CN_O is a set of unambiguously defined concepts (called *semantic concepts*) of IS engineering methods, A_O a set of associations (called *semantic associations*), and Φ: $CN_O \times A_O \rightarrow CN_O$ a function relating elements of CN_O with elements of CN_O through a semantic association that is an element of A_O. The ontology can be captured as a kind of semantic network. As mentioned before, we use the semantic concepts and semantic associations that have been defined in MDM, in order to organize our ontology. By using his system, for example, we can organize anchoring functions for Statechart shown in Figure 1 and Object Model in Figure 3 as follows.

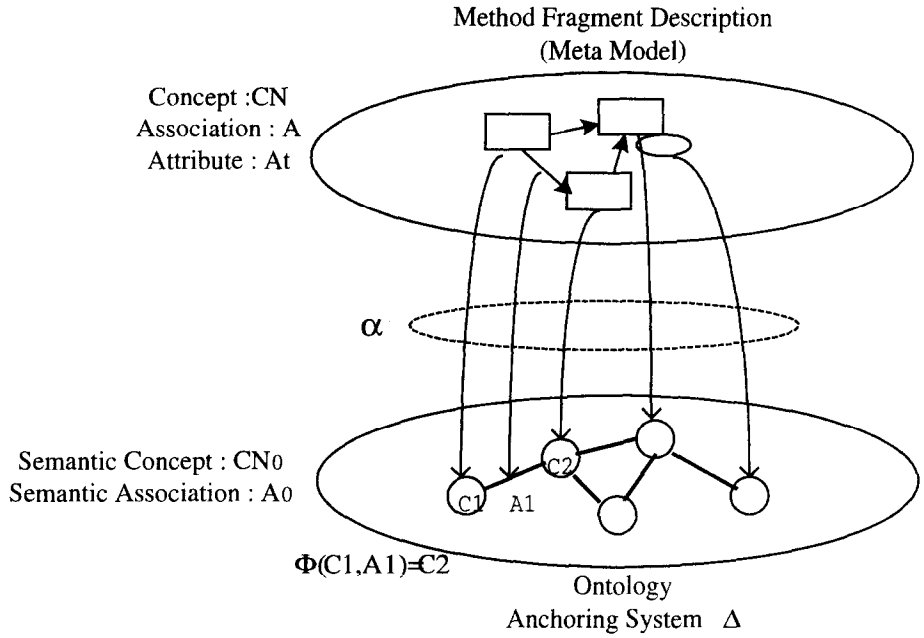

Fig. 2: Semantics of Method Fragments and Anchoring System

[Statechart]
α(State) = {State}, α(Transition) = {Transition}, α(Event) = {Event},
α(Firing Condition) = {Condition}, α(is source of) = {Change, Source},
α(is destination of) = {Change, Destination, Effect},
α(AND-decomposition) = {Description, Aggregation},
α(OR-decomposition) = {Description, Aggregation, Choice},
α(has-Event-Transition) = {TransitionTrigger},
α (has-Event-Firing Condition) = {Description}

[Object Model]
α(Object) = {Object, Activity, Actor}, α(Class) = {Object Class},
α(Attribute) = {Property, Data set},
α(Service) = {Function, Activity}, α(Association) = {Association},
α(has-Class-Object) = {Abstraction}, α(has-Class- Service) = {Capability, Manipulation},
α(has-Class-Attribute) = {Contents, Description}, α(participate in) = {Involvement}

Since the several associations with the same name occur, e.g. "has", we explicitly specify the concepts that participate in them in the above definitions, e.g. "has-Class-Object" (the association "has" between "Class" and "Object").

The Appendix A shows an extraction of MDM that is related to the above example and is sufficient to understand this paper. See [13] for a complete definition.

Anchoring systems formally capture the semantics of method fragments as much as possible. They prescribe the possible relationships between the elementary building blocks of method fragments, and they provide a uniform definition of these building blocks. It is important to notice that there still remains some non-formalisable part in the anchoring system: the definition of the concepts in natural language. The authors welcome suggestions to make progress on this boundary.

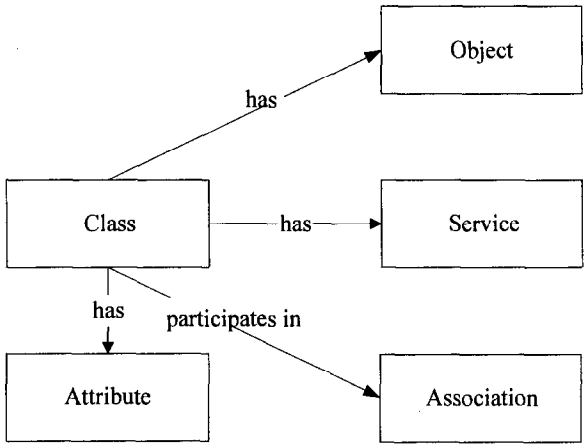

Fig. 3: Object Model Method Fragment

3. METHOD ASSEMBLY: EXAMPLE AND RULES

3.1. Method Assembly in the Product Perspective

In this section, we introduce a simple example of method assembly — assembling Object Model in Object-Oriented Analysis/Design and Statechart to Objectchart. Objectchart, proposed in [7], is an extension of Statechart to model reactive systems from an object-oriented view. Our framework of method assembly can explain how Objectchart was composed from the existing method fragments Object Model and Statechart.

The Object Model specifies a system as a set of objects communicating with each other. Objects have their specific attributes and change their values through inter-object communication. By sending messages to the other objects (or itself) an object requires of them (or itself) to provide the service that they (or it) encapsulate. The objects that are requested perform their service and may change their attribute values and/or return the computed results. Objects having the same attributes and services are modelled with a Class, which is a kind of template. Figure 3 shows the method fragment description of the Object Model at Diagram layer from conceptual level and product perspective.

Suppose now we have to produce Objectchart by assembling these two method fragments, i.e. the method models of Figures 1 and 3. Figure 4 shows the resulting method fragment of Objectchart in the same level, perspective and layer. As for this assembly process, we should note that the two method fragments belong to the same category in our three dimensional classification: conceptual level in abstraction, Diagram layer in granularity, and product in perspective. In addition we have product perspective of Objectchart in conceptual level and in Diagram Layer. Thus the method fragments with the same category can be assembled and we can get a new method with the same category.

The Statechart and Object Model are amalgamated to Objectchart by the following constructs:

1) A Class has a Statechart, which specifies its behaviour.
2) Attributes of a Class may be annotated to States in its Statechart. This indicates which attribute values are meaningful or visible in a specific state.
3) An Event issued during a Transition is a request of a Service to the other Object.
4) A Transition may change an Attribute value of an Object.

The first three constructions allow us to introduce new associations "has" between Class and State, "is annotated with", between Attribute and State, and "consist of". The concept Object participating in "consist of" stands for the object whose service is required, i.e. a receiver of the event. Furthermore, we employ the new concept "Post condition" for specifying the change of attribute value when a transition occurs. Therefore, post conditions can define the effect of service-execution on attributes.

Let's explore what manipulations were made and what kinds of constraints could be considered in this example. The basic manipulations that we applied here are:

1) Addition of a new concept (Post condition),
2) Addition of a new association (is_annotated_with, consists_of, has),
3) Addition of a new property (is_hidden).

First of all, when we assemble two method fragments, we should introduce at least one new concept or association. If we did not introduce anything, it would mean that a method fragment was completely included in another one. This case might be meaningless because we could not find the effect of this method assembly and the result was the same as the containing method fragment. This applies for the meaningless example of assembling ERD and Object Class Diagram (the super class of ERD), which we mentioned in Section 1. Furthermore, at least one connection between the two method fragments through newly introduced associations and/or concepts should be introduced, because the two method fragments are to be conceptually connected by the method assembly. Consequently, these constraints can be generalized as

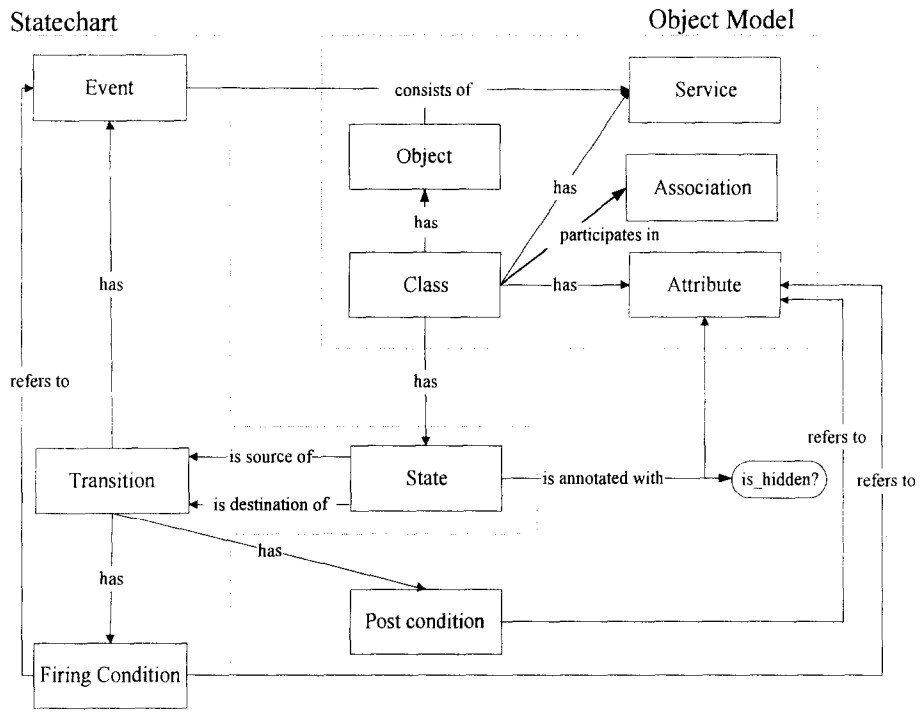

Fig. 4: Objectchart: Method Assembly in the Product Perspective

Rule 1) At least one concept, association or property should be newly introduced to each method fragment to be assembled, i.e. a method fragment to be assembled should not be a subset of another.

Rule 2) We should have at least one concept and/or association that connects between two method fragments to be assembled.

Rule 3) If we add new concepts, they should be connectors to both of the assembled method fragments.

Rule 4) If we add new associations, the two method fragments to be assembled should participate in them.

Moreover, the newly introduced associations should be meaningful, i.e.: only concepts that are allowed to be connected, can be connected. To determine this, semantics of concepts has to be known, which can be achieved by using an ontology (see Section 3.4).

The following additional rules can easily be determined, whose explanation we omit.

Rule 5) There are no isolated parts in the resulting method fragments.

Rule 6) There are no concepts which have the same name and which have the different occurrences in a method description.

These rules apply for method fragments in the conceptual level and diagram layer. If the method fragment to be assembled is related to the other levels or layers, the effect of assembly propagates to the others. It means that we should have the other types of rules. For example, the different concepts on the conceptual level should have different representation forms (notation) on the technical level. We will discuss a more elaborated style of rules and their formalization in Section 4.

3.2. Method Assembly in the Process Perspective

In the previous example, we illustrated product-perspective method assembly. Next, we turn to discuss the process-perspective method assembly also with the help of an example. Suppose we have the process descriptions for Object Model and for Statechart in Diagram layer at our disposal, e.g. for Object Model:

Draw an Object Model
- O1) Identify objects and classes,
- O2) Identify relationships,
- O3) Identify attributes and services

and for Statechart:

Draw a Statechart
- S1) Identify states,
- S2) Identify state changes and their triggers,
- S3) Cluster states, and so on.

According to [7], the recommended procedure for modelling Objectcharts is as follows:

Draw an Objectchart
- OC1) Draw an Object Model,
- OC2) For each significant class, Draw a Statechart, and
- OC3) Refine the Statechart to an Objectchart by adding post conditions and annotating states of the Statechart with attributes.

This procedure is constructed from the two process method fragments, Object Model (step OC1)) and Statechart (step OC2)) and seems to be natural. In more detail, between steps OC1) and OC2), we find that we should perform the activity of identifying the relationship "has" between Class and State shown in the Figure 4. The concept "Post condition" and its associations, say "refer to", and the association "is annotated with" are identified while the step OC3) is being performed. It means that newly added concepts and associations to connect the product-perspective method fragments to be assembled should not be identified until the associated concepts are identified. In fact, it is difficult for us to identify the association "has" between classes and states before we have identified classes or identified states and we should avoid this execution order of the activities (see also Figure 5).

Rule 7) The activity of identifying the added concepts and associations that are newly introduced for method assembly should be performed after their associated concepts are identified.

The rule mentioned above provides a criterion to make meaningful and useful procedures from manipulations on concepts and associations in Diagram Layer. Similarly, we can easily have the rule : we should not identify any associations until we identify their associated concepts in Diagram Layer. So the first step of method procedure should be identifying some concepts. This results from the natural execution order of human perception.

Another type of rules relates to the input/output order of products to activities. For example, the activity step O2) in Object Model consumes the identified objects and classes as its inputs that are produced by the step O1). The point in method assembly processes is what input-output relationships are added and/or changed. In this example, as shown in Figure 4, the step OC2) in Objectchart, which resulted from steps S1), S2) and S3) in Statechart, should consume the identified classes as its inputs. They are the output of the step O1) in Object Model, i.e. another method fragment. Therefore we can have the following rule:

Rule 8) Let A and B be the two method fragments to be assembled, and C the new method fragment. In C, we should have at least one product which is the output of A and which is the input of B, or the other way round.

This rule means that either of the method fragments to be assembled, say A, should produce input to the activities of B in the new method C. More examples of method assembly rules in process perspective will be shown in Section 4.

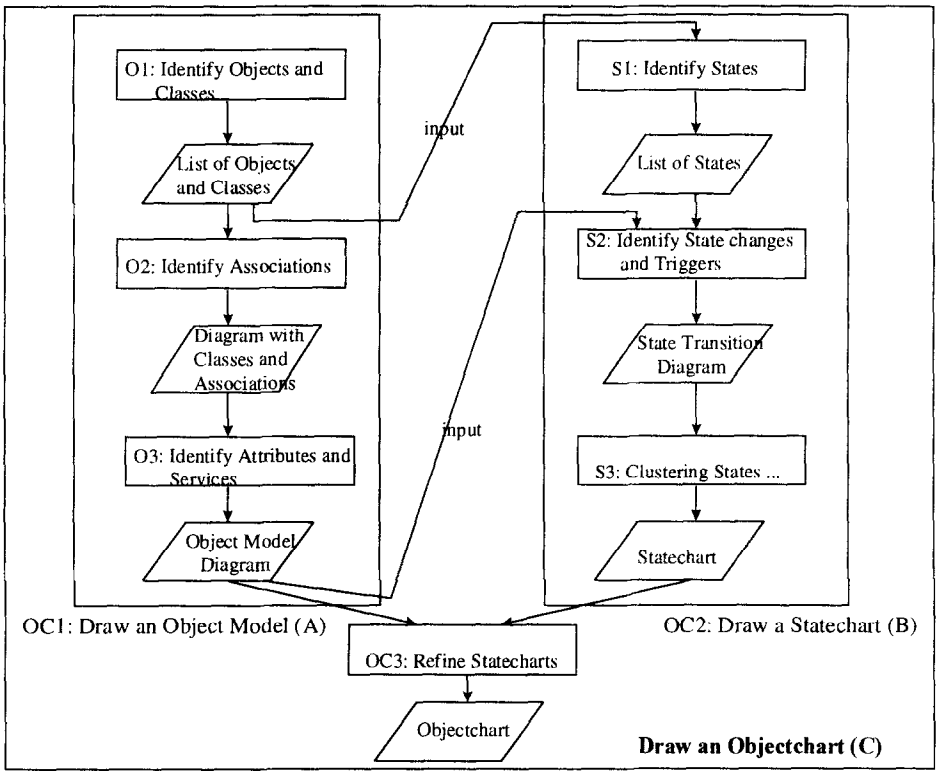

Fig. 5: Method Assembly in the Process Perspective

3.3. Discussion of Method Assembly on Three Dimensions

As we have shown in Section 2, method fragments can be considered on three dimensions: perspective, abstraction level and granularity layer. These dimensions can be used to improve, speed up, and simplify the method assembly process. We illustrate this with the following example. Assembling Object Model and Statechart, which are *product* fragments at the *Diagram* layer and at the *conceptual* level, implies the assembly of method fragments addressing the other perspective, abstraction level, and granularity layers. Associated with the Statechart and Object Model *product* fragments are modeling procedures, i.e. *process* fragments. The assembled modeling procedure results from the components of each of these two process fragments. Some of the rules that apply are:

Rule 9) Each product fragment should be produced by a "corresponding" process fragment.

Rule 10) Suppose a product fragment has been assembled. The process fragment that produces this product fragment consists of the process fragments that produce the components of the product fragment.

Also associated with the conceptual method fragments mentioned above are *technical* method fragments, such as Object Model and Statechart diagram editors, a repository to store object models and Statecharts, and a process manager to support the modeling procedures for object models and Statecharts. Similarly, the assembly of these technical method fragments results from the assembly of the corresponding conceptual method fragments:

Rule 11) A technical method fragment should supports a conceptual method fragment.

The assembly of fragments at the Diagram layer has also implications for the components of these fragments, which are at the Concept layer. In general, assembly of two method fragments results in the assembly of method fragments of lower granularity layers. As we have seen in Section 3.1, the assembly of Object Model and Statechart results in the assembly of Service and Event, Class and State, and Attribute and Firing Condition. A rule that applies to this is:

Rule 12) If an association exists between two product fragments, there should exist at least one association between their respective components

We have taken in the above example the assembly of conceptual product fragments at the Diagram layer as a starting point. However, the starting point can be at any combination of perspective, abstraction level, and granularity layer. Obviously, whatever starting point is used, the result of one assembly action is a cascade of other actions within the three-dimensional framework.

3.4. Semantic Aspect of Method Assembly

Note that all of the rules mentioned in Sections 3.1, 3.2 and 3.3 are syntactical constraints on descriptions of method fragments written in Entity Relationship Model or Flow Chart Model. To formalize actual method assembly processes more rigorously and precisely, we should consider some aspects of the meaning of method fragments. In the example of Objectchart, we associated the concept "Attribute" with "State". The question is in whatever method assembly we can always do it. The answer depends on the semantics of these concepts in the method fragments. Although how to specify the semantics of method fragments for method assembly is not an aim of this paper, we need to show how to combine with our assembly rules the existing technique for providing semantics such as ontology and anchoring systems. In Section 2.3, we introduced one of the techniques for providing formal semantics for method fragments, i.e. an anchoring system. We adopt the system that was proposed by Harmsen [13].

The anchoring system specifies which associations can connect to which concepts by using the function Φ. For example, the semantic concept "Object Class" cannot be associated with "Object" through any semantic association except for "Abstraction" (see Appendix). This constraint can help us to avoid introducing meaningless concepts and associations during constructing method fragments. Suppose that we try to add another association, say A, between "Class" and "Object" in the method fragment of Object Model shown in Figure 3. If the meaning of A is Abstraction", i.e. we set $\alpha(A) = \{\text{Abstraction}\}$, we can get a meaningful fragment as a result of the addition. If not, we could get the meaningless one, and it should not be allowable. Thus we can have the following rule for keeping semantic consistency (soundness).

Rule 13) There are no "meaningless" associations in product fragments, i.e. every association is "meaningful" in the sense that it can semantically consistently connect to specific concepts.

In this section, we outlined how to represent assembly constraints related to the meaning of method fragments. Figure 6 depicts the outline of the syntactical and semantic aspects of our assembly rules. The rules can be formally represented with first-order logical formulas as shown in Section 4. In the figure, we assemble two meta models MF#1 and MF#2 by connecting a new association A. Syntactic rules, e.g. rules 1) ~ 12), can be formalized as logical formulas that have the arguments MF#1, MF#2 and A, i.e.

PS(MF#1, MF#2, A) where PS is a predicate symbol. Semantic rules such as Rule 13) can be defined with the logical formulas that the semantic concepts and associations, i.e. α(MF#1), α(MF#2) and α(A) appear in. In the semantic rule of the figure, PM is a predicate symbol whose arguments are both the syntactic components (MF#1, MF#2 and A) and their semantic ones (α(MF#1), α(MF#2) and α(A)). Semantic rules specify the constraints on syntactic and semantic components of method fragments.

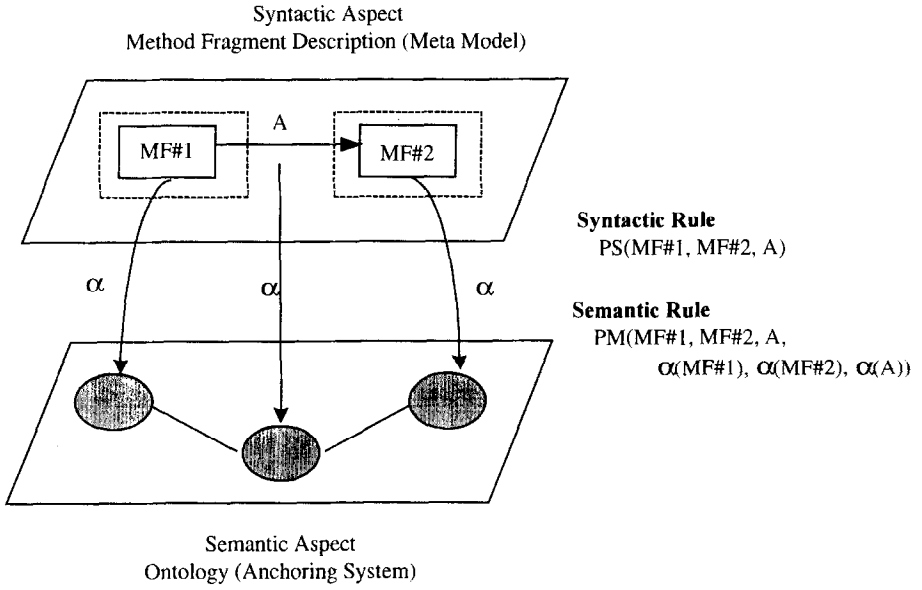

Fig. 6: Syntactic Rules and Semantic Rules for Method Assembly

4. METHOD ASSEMBLY: GUIDELINE AND FORMALIZATION

4.1. Requirements for Method Assembly

Method assembly should ensure that the selected method fragments are mutually adjusted, i.e. they have to be combined in such a way that the resulting situational method does not contain any defects or inconsistencies. Several types of defects can appear:

- Internal incompleteness, which is the case if a method fragment requires another method fragment that is not present in the situational method. For instance, a data model has been selected without the corresponding modelling procedure and tool.
- Inconsistency, which is the case if the selection of a method fragment contradicts the selection of another method fragment. For instance, two similar data modelling techniques have been selected without any additional reason.
- Inapplicability, which is the case if method fragments cannot be applied by project members, due to insufficient capability.

All these issues relate to the *internal* or *situation-independent quality* [14] of a situational method, i.e. the quality of a method without taking into consideration the situation in which the method is applied. The two most important criteria are:

- Completeness: the situational method contains all the method fragments that are referred to by other fragments in the situational method.
- Consistency: all activities, products, tools and people plus their -mutual- relationships in a situational method do not contain any contradiction and are thus mutually consistent.

Furthermore, we distinguish the following *method internal quality* criteria that are not treated in this paper for the sake of brevity and their details is in [13]:

- Efficiency: the method can be performed at minimal cost and effort
- Reliability: the method is semantically correct and meaningful
- Applicability: the developers are able to apply the situational method

The effort to achieve situation-independent quality of method fragments is considerable. Method fragments can be combined in a lot of ways, many of which are meaningless. Moreover, method fragments require other method fragments to be meaningful in a situational method, or require certain skills from the actors related to them. This is illustrated by the following small example. Suppose a process perspective method fragment *Draw an Object Model* (shown in Section 3.2) has been selected. The following should be at least verified ;

1) No similar method fragment already exists in the situational method,
2) The specification of the *Object Model* produced by the process fragment is selected,
3) Actors have the expertise to deal with this process fragment, and
4) The products required are produced by preceding selected process fragments (See also the examples in Section 3.1 and Section 3.2).

Internal method quality can only be achieved by a set of guidelines on the Method Engineering level. These formalized guidelines are presented in the form of axioms, which can be considered as an extension of the set of axioms, corollaries and theorems presented in Section 4. The axioms are grouped by the various quality criteria.

4.2. Classification of Method Assembly

In this section, the general internal quality requirements completeness and consistency are further partitioned by means of the three-dimensional classification framework.

Completeness is partitioned into:

- Input/output completeness, stating that if a process fragment requiring or manipulating a product fragment is selected, then that product fragment should be available in the situational method. Input/output completeness applies to the interaction of the two perspectives.
- Content completeness, stating that if a method fragment is selected, all of its contents have to be available too. Contents completeness applies to the relationship between granularity layers.
- Process completeness, requiring that all product fragments have to be, in some way, produced. Process completeness is related to the interaction of the two perspectives.
- Association completeness, requiring that product fragments on certain layers are always involved in an association, and that associations always involve product fragments. Association completeness relates to the product perspective.
- Support completeness, requiring that technical method fragments support conceptual method fragments. Support completeness applies to the relationship between abstraction levels.

Consistency is partitioned into:

- Precedence consistency, requiring that product fragments and process fragments are placed in the right order in the situational method. This type of consistency applies to the interaction between perspectives.
- Perspective consistency, requiring that the contents of product fragments is consistent with the contents of process fragments. Perspective consistency also applies to the interaction between perspectives.

- Support consistency, requiring that technical method fragments are mutually consistent. Support consistency relates to the relationships of technical method fragments.
- Granularity consistency, which imposes that the granularity layers of related method fragments are similar, and that their contents are mutually consistent. This type of consistency applies to the interaction between granularity layers.
- Concurrence consistency, which requires parallel activities to be properly synchronized. Concurrence consistency relates to the interaction of process fragments.
- Soundness (Semantic consistency) is the requirement that the situational method is semantically correct and meaningful.

In the next sub-section, each of these categories will be elaborated by means of an example taken from the Objectchart example.

4.3. Method Assembly Rules

4.3.1. Some Definitions

As noticed before, the natural language representation of method assembly rules creates some problems regarding ambiguity and implementability. Therefore we have formalized our theory regarding method fragments, and expressed the rules in that formalization. In this sub-section, we only show the part of the formalization required in the context of this paper. Moreover, we give examples of rules, of which some can be formalized well.
The formalization employs the following notions:

- *Set*, which represents a category of similar method fragments.
- *Predicate*, which represents a relationship between Method Base concepts.
- *Function*, which represents the assignment of the method fragment properties to method fragments
- The usual logical quantifiers and operators.
- The operators $<$, $=$, $\in$, $\subset$, $\cup$ and $\cap$.

The following sets are defined:

$M = C \cup T$, the set of method fragments

$C = R \cup P$, the set of conceptual method fragments: e.g. Draw an Object Model, Object Model, Statechart, Identify Objects and Classes, Class, Object, Service, Transition "has" Event, List of States.

R the set of product fragments, e.g. Class, Object, Event, Object Model, Statechart, List of States

P the set of process fragments, e.g. Identify Objects and Classes, Identify Associations, Identify States, Draw an Object Model.

$CN \subseteq R$, the set of concepts, e.g. Class, Object, Service, State, Event.

$A \subseteq R$, the set of associations, e.g. Transition "has" Event, State "is annotated with" Attribute.

T the set of technical method fragments.

The following sets and function are defined for an anchoring system that provides semantics of method fragments. (see Section 2.3).

CN_O, the set of the semantic concepts.

A_O, the set of the semantic associations.

α: $M \rightarrow \wp(CN_O \cup A_O)$, the anchoring function.

If a method fragment is selected for inclusion in a situational method, it is indexed with an "*s*", for instance: R_s is the set of selected product fragments.
The following predicates are used in this section:

- *contents* and *contents** $\subseteq R \times R \cup P \times P$ to represent the non-transitive and transitive consists-of relationship between method fragments, e.g. *contents*(Class, Object Model);
- *manipulation* $\subseteq P \times R$, to represent the fact that a process fragment manipulates (i.e. produces, updates, etc.) a certain product fragment, e.g. *manipulation*(Draw an Objectchart, Objectchart);
- *involvement* $\subseteq A \times R$, to represent the fact that an association involves a product fragment , e.g. *involvement* (is annotated with, Objectchart);
- *prerequisite* $\subseteq P \times R$, to represent the fact that a process fragment requires a product fragment for its execution, e.g. *prerequisite*(Identify Associations, List of Classes and Objects);
- *precedence* $\subseteq P \times P$, denote the precedence relationship between process fragments, e.g. *precedence*(Identify Associations, Identify Classes and Objects);
- *support* $\subseteq C \times T$, to represent that a technical method fragment supports a conceptual method fragment, e.g. *support*(Statechart, STATEMATE);
- *concurrence*, to represent the fact that two process fragments can be performed in parallel, e.g. *concurrence*(Identify Associations(O2), Identify States(S1)) (see Figure 5).
- *layer*: $M \rightarrow \{Method, Stage, Model, Diagram, Concept\}$, to return the layer of the method fragment (see Section 2.2), e.g. *layer*(Objectchart)=Diagram, *layer*(Class)=Concept.

Below, each type of completeness and consistency, as defined in Section 4.1, is related to our Objectchart example. We assume that both Object Model, Statechart, and Objectchart should be part of a complete and consistent situational method, M_s.

4.3.2. Completeness Rules

Input/Output Completeness Step 2 of the Objectchart modeling procedure requires an Object Model. The description of the Object Model should therefore exist in the situational method. In general, the rule is: Required product fragments should have been selected for the method assembly , i.e.

$$\forall p \in P_S, r \in R[prerequisite(p,r) \rightarrow r \in R_S]$$

Contents Completeness Concepts (product fragments) such as Class, Object, State, Service, Transition etc. should always be part of another product fragment. Note that this is indeed the case, as they are all components of Statechart. In a formalized way, this rule is defined as follows:

$$\forall r_1 \in R_S \exists r_2 \in R_S[layer(r_1) = \text{concept} \rightarrow contents^*(r_2, r_1) \wedge layer(r_2) \in \{\text{Model, Diagram}\}]$$

Process Completeness Suppose the Objectchart is included in the situational method. Then it has to be produced by some process fragment that is also included. In general, selected product fragments at the lowest four granularity layers have to be produced by a selected process fragment, i.e.

$$\forall r \in R_S \exists p \in P_S[layer(r) \neq \text{Concept} \rightarrow manipulation(p,r)]$$

Association Completeness Suppose both the Object Model and Statechart have been selected for inclusion in the situational method. Then they should be connected by at least one association (note, again, that this is the case; they are connected by even more than one association). In general, if more than one diagram layer product fragment has been selected, diagram layer product fragments should be associated with at least one other diagram layer product fragment (Rule 4)).

$$\forall r_1, r_2 \in R_S \exists a \in A_S [layer(r_1) = \text{Diagram} \wedge layer(r_2) = \text{Diagram} \wedge r_1 \neq r_2 \rightarrow involvement(a, r_1) \wedge involvement(a, r_2)]$$

Also Rule 3) is an example of an association completeness rule:

$$\forall r_1, r_2 \in R_S \exists a_1, a_2 \in A_S \exists c \in CN_S [(layer(r_1) = \text{Diagram} \wedge layer(r_2) = \text{Diagram} \wedge r_1 \neq r_2) \rightarrow involvement(a_1, r_1) \wedge involvement(a_2, r_2) \wedge involvement(c, r_1) \wedge involvement(c, r_2)]$$

From these rules we can deduce, that Rule 2) is redundant.

Support Completeness Suppose the STATEMATE editor was selected for inclusion in our situational method. Then, the Statechart product fragment that is supported by this editor should also be included. In a formalized way, this rule, i.e. Rule 11) is defined as follows:

$$\forall t \in T_S, r \in R [support(r, t) \rightarrow r \in R]$$

4.3.3. Consistency Rule

Precedence Consistency In the modeling procedure for Objectchart, step OC2 requires an Object Model. This Object Model should be produced by a step *before* step OC2. In general: a process fragment producing a required product fragment should be placed before the process fragment requiring the product fragment, i.e.

$$\forall p_1 \in P_S, r \in R_S \exists p_2 \in P_S [prerequisite(p_1, r) \rightarrow manipulation(p_2, r) \wedge precedence(p_1, p_2)]$$

This rule is a part of Rule 7). This rule means that we should have at least one new process fragment and this new fragment should not be first in the order of the assembled process fragments.

In the example of Figure 4, we have a new process fragment "Refine Statechart (OC3)", and it cannot be performed before Draw an Objectchart and Draw a Statechart. The above rule specifies the latter part. We can also formalize the former part.

Perspective Consistency Objectchart is produced by the modeling procedure presented in Section 3.2. The components of Objectchart, its concepts, should be produced by components of this fragment. As a general rule: If a product fragment is produced by a certain process fragment, then all of its contents should be produced by the sub-processes of that process fragment, i.e.:

$$\forall p_1, p_2 \in P_S, r \in R_S, b \in B \exists r_2 \in R_S [manipulation(p_1, r_1) \wedge contents(p_1, p_2) \rightarrow contents(r_1, r_2) \wedge manipulation(p_2, r_2)]$$

Granularity Consistency An example of a granularity consistency rule is Rule 12) (Section 3.4), stating that if two product fragments are associated, there should be at least an association at the Concept layer in their perspective contents as well, i.e.:

$$\forall a_1 \in A_S, r_1, r_2 \in R_S, l_1, l_2 \in L \exists c_1, c_2 \in CN_S, a_2 \in A_S [involvement(a_1, r_1) \wedge involvement(a_1, r_2) \rightarrow contents^*(r_1, c_1) \wedge contents^*(r_2, c_2) \wedge involvement(a_2, c_1) \wedge involvemnet(a_2, c_2)]$$

Concurrence Consistency Suppose the Objectchart process fragment consists, to speed up the process, of two steps that are concurrently executed. This may only be the case, if they do not require complete products from each other. So, for instance, steps OC1 and OC2 of the Draw an Objectchart fragment may

not be concurrently executed, as step OC2 required some intermediate results produced by step OC1. However, within this fragment some steps can be performed concurrently, e.g. O2 and S1. The concurrence consistency rule is defined as follows:

$$\forall p_1, p_2 \in P_S, r \in R_S [concurrence(p_1, p_2) \rightarrow \neg(prerequisite(p_1, r) \wedge manipulation(p_2, r)) \wedge \neg(prerequisite(p_2, r) \wedge manipulation(p_1, r))]$$

Soundness (Semantic Consistency) In Rule 13), we stated that newly introduced associations should be meaningful. This can be translated into the rule, that for each association at the Concept layer of the situational method, there should be a corresponding association in the ontology between the respective semantic concepts involved:

$$\forall a_1 \in A,\ c_1, c_2 \in CN\ \exists d_1, d_2 \in CN_O,\ a_2 \in A_O [involvement(a_1, c_1) \wedge involvement(a_1, c_2) \rightarrow d_1 \in \alpha(c_1) \wedge d_2 \in \alpha(c_2) \wedge involvement(a_2, d_1) \wedge involvement(a_2, d_2)]$$

For example, we associated "Attribute" of Object Model with "State" of Statechart in the method assembly process, and introduced a new association "is annotated with", whose meaning is "Description". Since the anchor system, which provides the formal semantics, has an association between the semantic concepts "Data Set" (see Section 3.4, α(Attribute) = {Data Set, Property}) and "State" (α(State)={State}), it is easily deducible the above formula holds on the association "is annotated with". See Appendix A and check Φ(State, Description) = Data Set.

Remember that the aim of this paper is not to provide a complete anchoring system but to propose our assembly techniques based on patterns of rules. The above example can show that semantic information, which is provided by the existing anchoring system, can be directly and easily embedded to our rules.

5. RELATED WORK

As mentioned before, several meta-modelling techniques were proposed, e.g. they were based on Entity Relationship Model, Attribute Grammar, Predicate Logic and Quark Model. Comparison of meta-modelling techniques and their languages was also discussed in [12]. We choose the techniques that have the capability of method assembly and discuss the differences with our work.

Almost all approaches to meta-modelling are using Entity Relationship Model (ER). Some applied Predicate Logic to describing the properties, which cannot be represented with just the ER notation. For instance, the Viewpoints approach [18] combines ER and Predicate Logic. It aims at constructing a method with multiple views from the existing methods. In other words, we can define the assembly mechanism of the products, which are produced by the different existing methods. The approach also provides the function for defining constraints to maintain consistency on the products that are produced by the existing methods. However, the Viewpoint approach did not provide the capability of specifying constraints on method assembly, but focused on defining the constraints on product assembly.

UML (Unified Modeling Language) [28] has a function to define how combine nine UML diagrams using its meta model. However it does not suggest which combinations can produce meaningful usage of the diagrams, because it has no rules or guidelines of the meaningful combinations. Suppose that we generate the method where a developer is forced to complete a UML state diagram before starting writing a UML use-case diagram. The method does not seem to be so useful, because a use-case diagram is for extracting system behavior so as to write a state diagram. The meta model of UML cannot provide the guidelines that prevent us from generating this kind of meaningless method. Our precedence consistency rules in Section 4.3.3 do not allow us to perform this assembly of a use-case diagram and a state diagram.

Software Quark Model [1] tried to formalize a restricted set of atomic concepts, which can specify any kind of software products and it can be considered as a product perspective of meta-modelling. The aim of the model seems to be not method assembly in product level, but maintaining causality relationships among the software products produced in various stages of a software development cycle through atomic concepts.

In his article, Song investigated the existing integrated methods, into which several different methods were integrated, and classified method integration from benefit-oriented view, i.e. classification criteria are based on what benefit we can get by the integration [25]. He did not use the term *assembly* but *integration*. According to his classification, we can have two categories: function-driven (a new function is added) and quality-driven (the quality of a method is improved). He also classified these two categories in detail based on which components of methods are integrated, e.g. Artifact Model Integration, Process Integration, Representation Integration and so on. His work is a pioneer of method assembly research. However, he did not discuss how to integrate (assemble) methods or what rules should hold for each category but just classified the existing integration patterns. And, all of his proposed classes are not necessary orthogonal, i.e. an integration is included in several classes. Furthermore our classification includes Song's classification. Figure 3 is an example of Song's Artifact Model Integration, i.e. method assembly in Conceptual Level, Product Perspective and Diagram Layer.

And none of the meta-modelling and assembly (integration) techniques mentioned above considered the semantic aspects of method fragments.

General work regarding *anchoring systems* or *ontology* can be found in [7] and some of the best known academic approaches in method engineering field, especially for product fragments are in [2, 19, 27, 29]. The CRIS framework [19] consists of method components and their relationships used in various stages of IS development. Sowa and Zachman [27] developed the Information Systems Architecture (ISA) framework consisting of roles (for instance planner or builder) and perspectives (for instance the process perspective, the network perspective). *European Modelling Language* (EML) [2] consists of a *Concepts Model*, which contains elementary IS engineering concepts, their attributes, and their relationships, and is intended to provide a default standard method specification language for co-operative IS projects within the European market. The differences among these work are on which basic semantic concepts, i.e. common terminology were selected and adopted. No anchoring system mentioned above has been found identifying the basic processes in IS engineering. Moreover, they seem to have been defined very pragmatically. We considered that the above problems have been improved in MDM [13]. For example, MDM includes basic semantic concepts for process fragments in addition to product fragments, and the concepts have been extracted from a wider spectrum of IS. Although in our method assembly rules we adopted the MDM for the above reasons, the rules are essentially independent on which anchoring system we select.

6. CONCLUSION AND FUTURE WORK

This paper clarifies how to assemble method fragments into a situational method and formalize rules to construct meaningful methods. We have already extracted over 80 rules thought real method assembly processes. Our rules are general ones which are applicable for arbitrary method assembly, and we may need some rules for specific kinds of method assembly. These rules probably include semantic information on method fragments and on systems to be developed. Our next goal is to assess our generic rules in more complicated and larger-scale assembly processes, e.g. whether our rules are sufficient and minimal to specify method assembly processes as general rules, and to look for specific rules as method assembly knowledge.

Our rules are described with predicate logic, so we have a possibility to check method fragments automatically during the assembly processes. To get efficient support, we should consider how our rules can be efficiently executed in our method base system, which stores various kinds of method fragments. As reported elsewhere, we are currently developing the Computer Aided Method Engineering (CAME) tool, called Decamerone [11], which includes a comprehensive method base system. A support function for method assembly processes based on our assembly rules is currently under development. Functionality for adaptive repository generation and customisable process managers is being realised. Next to this, the Method Engineering Language (MEL) is under development [12]. This language allows us to describe method fragments from the various relevant dimensions. Operators for the manipulation, storage and retrieval of method fragments in the method base have been defined. To clarify which method fragments are suitable and useful for a specific situation is one of the most important research issues and empirical studies are necessary such as [16], [23] and [24], in addition to the issue on complete formal semantics of methods and method fragments. In particular, according to [23] that discusses human cognitive limitation, human developers tend to pay more attention to syntactical aspects of products rather than semantic ones in information systems modeling processes. This leads to the significance of the

methods that match the semantics of the problem domains of modeling, in order to prevent developers from neglecting the semantic aspects of products in modeling processes.

Acknowledgements — The authors would like to thank the anonymous reviewers for their insightful comments to improve the earlier version of this paper.

REFERENCES

[1] T. Ajisaka. The software quark model: a universal model for CASE repositories. In *Journal of Information and Software Technology*, **38**(3):173–180 (1996).

[2] ASSET: ESPRIT Project N.7703, EML Concepts Definition & Work Products. *Report on work in progress V 1.4*, Siemens-Nixdorf Informationssysteme AG/EMSC (1994).

[3] S. Brinkkemper. *Formalisation of Information Systems Modelling*. Thesis Publisher, Amsterdam (1990).

[4] S. Brinkkemper. Method engineering: engineering of information systems development methods and tools. In *Journal of Information and Software Technology*, **38**(4):275–280 (1996).

[5] S. Brinkkemper, K. Lyytinen and R. Welke, editors. *Method Engineering: Principles of Method Construction and Tool Support*. Chapman & Hall (1996).

[6] M. Bunge. *Treatise on Basic Philosophy, Vol. 3, Ontology I: The Furniture of the World*, D. Reidel Publishing Company, Dordrecht (1977).

[7] F. Coleman, F. Hayes and S. Bear. Introducing objectcharts or how to use statecharts on object-oriented design. *IEEE Trans Soft. Eng.*, **18**(1):9–18 (1992).

[8] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press (1978).

[9] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shutull-Trauring and M. Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Trans. Soft. Eng.*, **16**(4):403–414 (1990).

[10] F. Harmsen, S. Brinkkemper and H. Oei. Situational method engineering for information system projects. In *Methods and Associated Tools for the Information Systems Life Cycle, Proceedings of the IFIP WG8.1 Working Conference CRIS'94*. T. Olle and A. Verrijn Stuart, eEditors, pp. 169–194, North-Holland, Amsterdam (1994).

[11] F. Harmsen and S. Brinkkemper. Design and implementation of a method base management system for a situational CASE environment. In: *Proceedings of the APSEC'95 Conference*, pp.430–438, IEEE Computer Society Press, Los Alamitos, CA (1995).

[12] F. Harmsen and M. Saeki. Comparison of four method engineering languages. In *Method Engineering: Principles of Method Construction and Tool Support*, S. Brinkkemper, K. Lyytinen and R. Welke, editors, Chapman & Hall, pp.209–231 (1996).

[13] F. Harmsen. *Situational Method Engineering*. Moret Ernst & Young (1997).

[14] R. van de Hoef and F. Harmsen. Quality requirements for situational methods. In *Proceedings of the Sixth Workshop on the Next Generation of CASE Tools*, G. Grosz, editor, Jyväskylä, Finland (1995).

[15] T. Katayama. A hierarchical and functional software process description and its enaction. In *Proceedings of 11th Int. Conf. on Software Engineering*, pp.343–352, IEEE Computer Society Press, Baltimore (1989).

[16] M. Klooster, S. Brinkkemper, F. Harmsen, and G. Wijers. Intranet Facilitated Knowledge Management: A theory and tool for defining situational methods. In *Proceedings of CAiSE'97. Lecture Notes in Computer Science 1250*, A. Olive and J.A. Pastor, editors, Barcelona, pp.303–317, Springer Verlag (1997).

[17] J. Martin. *Information Engineering, Book II – Planning and Analysis*. Prentice Hall, Englewood Cliffs (1990).

[18] B. Nuseibeh, J Kramer, and A. Finkelstein. Expressing the relationship between multiple view in requirements specification. In *Proceedings of 15th Int. Conf. on Software Engineering*, Baltimore, IEEE Computer Society Press, pp. 187–197 (1993).

[19] T.W. Olle, J. Hagelstein, I.G. MacDonald, C. Rolland, H.G. Sol, F.J.M. van Asssche, and A.A. Verrijn-Stuart. *Information Systems Methodologies - A Framework for Understanding*, 2nd Edition. Addison-Wesley (1991).

[20] J. Rumbaugh, M.Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (1991).

[21] M. Saeki and K. Wen-yin. Specifying software specification and design methods. *In Proceedings of CAiSE'94, Lecture Notes in Computer Science 811*, G. Wijers, S. Brinkkemper, T. Wasserman, editors, Springer Verlag, pp. 353–366, Berlin (1994).

[22] M. Saeki. Are methods really useful?. In *Proceedings of 20th Int. Conf. on Software Enginnering Volume II*, Kyoto, pp.18, IEEE Computer Society Press (1998).

[23] K. Siau, Y. Wand, and I. Benbasat. The relative importance of structural constraints and surface semantics in information modeling. In *Information Systems Journal*, **22**(2&3):155–170 (1997).

[24] K. Slooten and B. Hodes. Characterizing IS development projects. In *Method Engineering: Principles of Method Construction and Tool Support*, S. Brinkkemper, K. Lyytinen and R. Welke, editors, pp.29–44, Chapman & Hall (1996).

[25] X. Song. A framework for understanding the integration of design methodologies. In *ACM SIGSOFT Software Engineering Notes*, **20**(1):46–54 (1995).

[26] P.G. Sorenson, J.P. Tremblay, and A.J. McAllister. The metaview system for many specifications environements. In *IEEE Software*, **30**(3):30–38 (1988).

[27] J.F. Sowa and J.A. Zachman. Extending and formalizing the framework for information systems architecture. In *IBM Systems Journal*, **31**(3):590–616 (1992).

[28] *UML Summary Version 1.1*. Rational Software Corp. (1997).

[29] Y. Wand. Ontology as a foundation for meta-modelling and method engineering. In *Journal of Information and Software Technology*, **38**(4):281-288 (1996).

[30] P. Ward and S. Mellor. *Structured Development for Real-time Systems*. Yourdon Press (1985).

APPENDIX A: DEFINITIONS OF THE METHODOLOGY DATA MODEL

This appendix is a part of the definition of the anchoring system Methodology Data Model (MDM) [13] so that the readers can understand this paper. See [13] if the complete definition is necessary for further understanding the MDM.

Definition 1 The MDM consists of concepts, associations, and relationships between concepts and associations. The following definition therefore addresses a set of concepts, a set of associations, and a function relating elements of the two sets. The *Methodology Data Model* is defined as a structure $\Delta_{MDM} = \langle CN_O, A_O, \Phi \rangle$, with

- CN_O, the set of MDM concepts,
- A_O, the set of MDM associations, and
- Φ: $CN_O \times A_O \rightarrow CN_O$, a function mapping MDM concepts and MDM associations on MDM concepts.

Definition 2 CN_O = {Activity, Actor, Association, Condition, Data Set, Event, Function, Object, Object Class, Property, State, Transition, ...}, where

- **Activity** is a step to perform a *function* in an information *system*.
- **Actor** is a person or machine performing an *activity*.
- **Association** is a relation between *object classes* in specific *roles*.
- **Condition** is the dependency upon which a *transition* can fire or a *decision* can be taken.
- **Data Set** is a stable, aggregated collection of data described by *objects* or *object classes* and used by a *data flow*.
- **Event** is a significant occurrence inside or outside the *system*, influencing the behaviour of *activities*.
- **Function** is a specific occupation within a *system*.
- **Object** is a material or abstract thing within a *system* with one or more *properties*, which can have a *state* and a behaviour represented by an *activity*.
- **Object Class** is an abstracted category of *objects*.
- **Property** is a quality of an *object class*.
- **State** is the mode of being or condition of an *activity* or an *object class*.
- **Transition** is a trigger, caused by the end of the execution of an *activity*, an *event*, or the fulfilment of a *condition*, resulting in a *state* change or the start of the execution of an *activity*.
- ...

Definition 3 A_O = {Abstraction, Aggregation, Capability, Change, Choice, Contents, Description, Destination, Effect, Involvement, Manipulation, Source, TransitionTrigger, Trigger, ...}, where

- **Abstraction** is the categorisation of *objects* into an *object class*.
- **Capability** is the ability of an *actor* to perform an *activity* or a *function*.
- **Change** is the effect of a *transition*.
- **Choice** is the solution for a *problem*.
- **Contents** is the description of a *data set* by *object classes*.
- **Description** is the characterisation of a concept instance by another concept instance.
- **Destination** is the destination of a movement or flow.
- **Effect** is the influence of one concept instance on another.

- **Involvement** is the inclusion of a concept instance in an *association.*
- **Manipulation** is the way in which an *activity* describes the behaviour of an *object.*
- **Source** is the source of a movement or flow.
- **TransitionTrigger** is the execution of a *transition* by an *event.*
- **Trigger** is the initiation of an *activity* by an *event.*
- ...

Definition 4 Φ: $CN_O \times A_O \rightarrow CN_O$, the partial function mapping MDM concepts and MDM associations on MDM concepts, is defined as follows:

Φ(Activity, Capability) = Actor, Φ(Activity, Effect) = Transition,
Φ(Activity, Manipulation) = Object, Φ(Activity, Trigger) = Transition,
Φ(Actor, Capability) = Function, Φ(Attribute, Description) = Object Class,
Φ(Event, TransitionTrigger) = Transition, Φ(Event, Trigger) = Activity,
Φ(Object Class, Abstraction) = Object, Φ(Object Class, Aggregation) = Data Set,
Φ(Object Class, Involvement) = Association, Φ(Object, Aggregation) = Data Set,
Φ(State, Description) = Data Set, Φ(State, Description) = Activity,
Φ(Transition, Change) = State, Φ(Transition, Description) = Activity,
...

Φ describes a semantic network of MDM concepts and associations. Due to the complexity of this network, we have chosen not to use a graphical notation, which would result in an incomprehensible figure.