Assembly Techniques for Method Engineering

Sjaak Brinkkemper[1]***, Motoshi Saeki***[2]***, Frank Harmsen***[3]

[1]*Baan Company R & D, P.O. Box 143, 3770 AC Barneveld, the Netherlands, sbrinkkemper@baan.nl*
[2]*Tokyo Institute of Technology, Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan, saeki@cs.titech.ac.jp*
[3]*Moret Ernst & Young, P.O. Box 3101, 3502 GC Utrecht, the Netherlands, nlharms4@mey.nl*

Abstract. As projects for developing information systems are getting larger and more complicated, we need to have more advanced development methods suitable for every development situation. Method engineering is the discipline to construct new 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 presented in first order predicate logic and play an important role in the assembly process of 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.

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 one of the most significant key factors to achieve great success of development projects. Until now, many methods such as structured analysis/design [De Marco 78] and object-oriented analysis/design [Rumbaugh91] 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 includes some parts, which do not fit to the characteristics of real projects and their contexts. 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 [Harmsen 94, Brinkkemper 96] 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 [Rumbaugh 91] 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 produced a powerful new method that could model complicated systems from multiple viewpoints: object view, behavioural view and functional view. Therefore, method assembly is a significant technique to construct both situational methods and powerful methods with multiple viewpoints.

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 meaningless methods.

Several researchers applied very adequate meta-modelling techniques based on Entity Relationship Model [Brinkkemper 91, Sorenson 88, Nuseibeh 95], Attribute Grammars [Katayama 89, Song 94], Predicate Logic [Brinkkemper 91, Saeki 94, Nuseibeh 95] and Quark Model [Ajisaka 96] 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 method assembly function itself yet. Song investigated existing methods, such as OMT and Ward/Mellor's Real Time SDM [Ward 85], and classified the way various methods are put together [Song 95]. 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. Section 3 presents method assembly by using 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. We discuss these guidelines and constraints, and their formalization in section 4. Sections 5 and 6 summarize related work and our work respectively.

2 A Classification Framework for Method Fragments

2.1 Method Fragments

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 [Harel 90]. 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 Fig. 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.)

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.

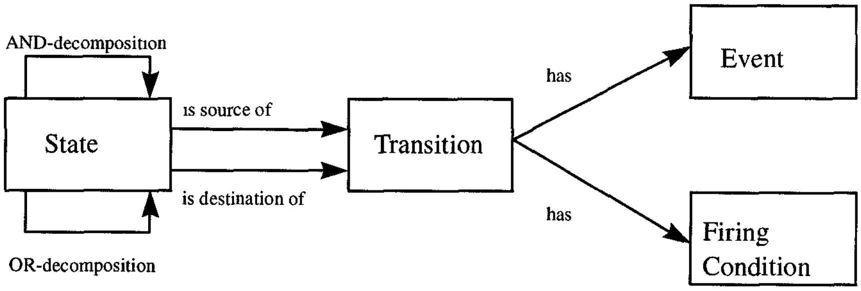

Fig.1 Statechart Method Fragment

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. Fig.1 is a description of the product perspective.

The abstraction dimension constitutes of 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 is shown in Fig. 1, whereas the corresponding technical fragment is the STATEMATE tool for specifying Statecharts [Harel 90].

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 [Brinkkemper 96] 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* [Martin 90] stage.
- **Model**, which addresses a perspective [Olle 91] 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*

3 Method Assembly Technique

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 [Coleman 91], 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) encapsulatedly have. 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. Fig. 2 shows the method fragment description of the Object Model at Diagram layer from conceptual level and product perspective.

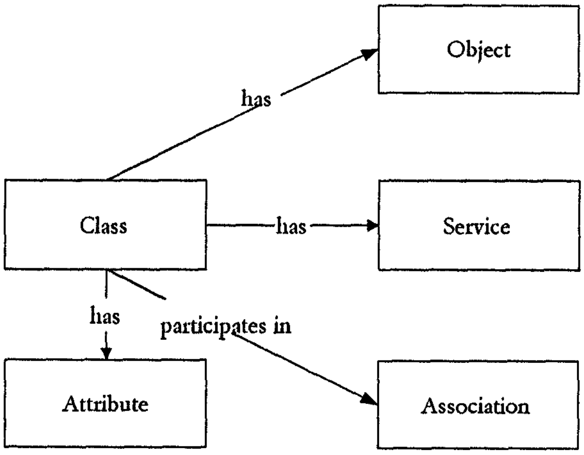

Fig.2 Object Model Method Fragment

Suppose now we have to produce Objectchart by assembling these two method fragments i.e. the method models of Figs. 1 and 2. Fig. 3 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.

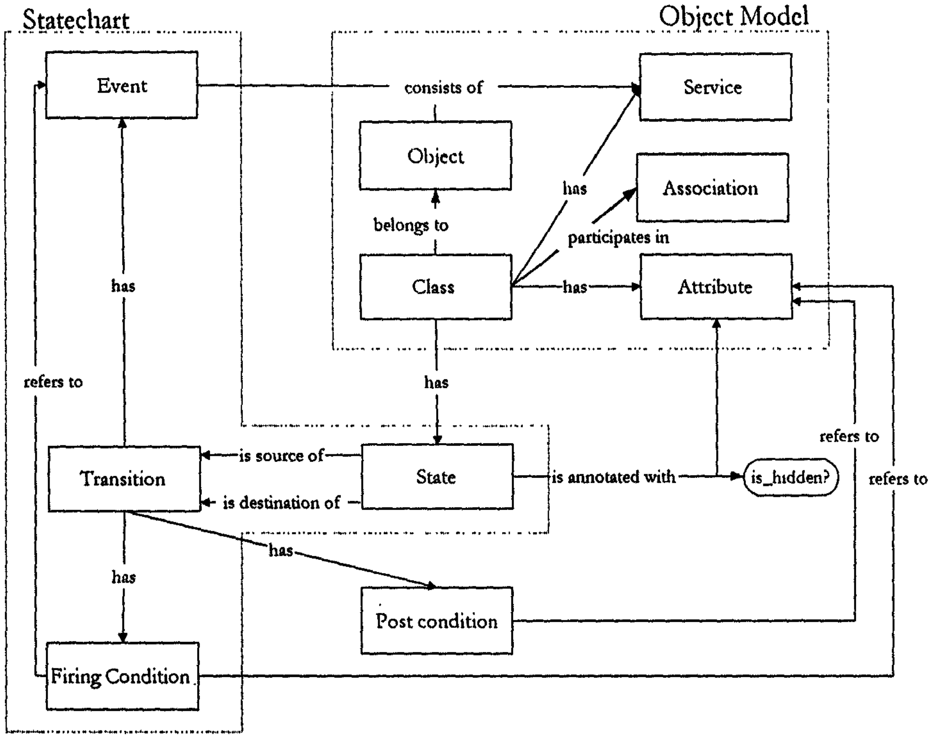

Fig. 3 Objectchart : Method Assembly in the Product Perspective

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

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 "consists of" . The concept Object participating in "consist of" stands for the object of which a 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

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.

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 [Coleman 92], 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 Fig. 3. The concept "Post condition" and its associations, say "refers 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 Fig. 4).

Rule 7) The activity of identifying the added concepts and relationships 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.

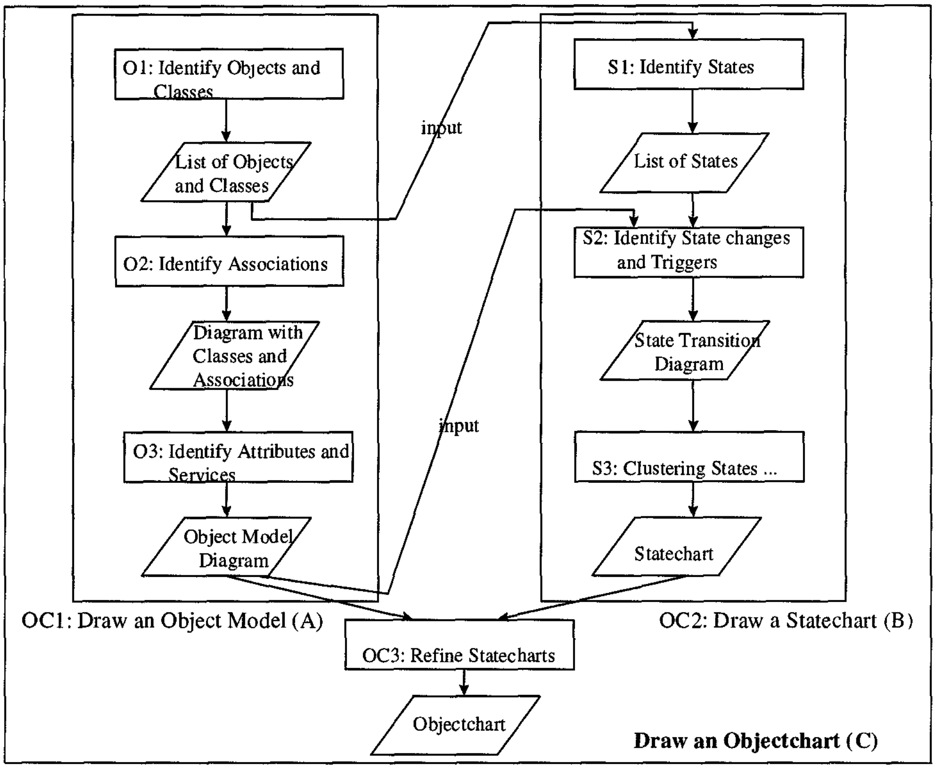
O1: Identify Objects and Classes
List of Objects and Classes
O2: Identify Associations
Diagram with Classes and Associations
O3: Identify Attributes and Services
Object Model Diagram
OC1: Draw an Object Model (A)
input
input
S1: Identify States
List of States
S2: Identify State changes and Triggers
State Transition Diagram
S3: Clustering States ...
Statechart
OC2: Draw a Statechart (B)
OC3: Refine Statecharts
Objectchart
Draw an Objectchart (C)

Fig. 4 Method Assembly in the Process Perspective

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 which 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 Fig. 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.

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.

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* [Hoef 95] 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 [Harmsen 97]:

- 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 sect. 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 sect. 3.1 and sect. 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 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.

Note that our concepts of "completeness" and "consistency" are syntactical constraints on descriptions of method fragments written in Entity Relationship 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. How to specify the semantics of method fragments for method assembly is one of the most important and interesting future topics.

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

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, some of which are 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 Classes and Objects, Class, Object, Service, Transition "has" Event, List of States.

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

P the set of process fragments, e.g. Class, Object, Service, State, Event.

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

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

T the set of technical method fragments.

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 Fig.4).
- *layer*: $M \rightarrow$ *{Method, Stage, Model, Diagram, Concept}*,to return the layer of the method fragment (see sect. 2.2), e.g. *layer*(Objectchart)=Diagram, *layer*(Class)=Concept.

Below, each type of completeness and consistency, as defined in sect. 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.

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

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:

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

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.

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

Association completeness

Suppose both the Object Model and State Chart 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 Fig. 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:

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

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 [Harmsen 96]. We pick up a few representatives and discuss their relevance to 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 [Nuseibeh 92] 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, it discusses about the constraints on the assembled products but not constraints on method assembly processes themselves.

Software Quark Model [Ajisaka 96] 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 is based on what benefit we can get by the integration [Song 95]. 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. Our framework is completely orthogonal and we have shown some guidelines and rules to produce meaningful methods. Furthermore our classification includes Song's classification. Fig. 3 is an example of Song's Artifact Model Integration, i.e. method assembly in Conceptual Level, Product Perspective and Diagram Layer.

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 [Harmsen 95], 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 [Harmsen 96]. 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 [Slooten 96] and [Klooster 97].

References

[Ajisaka 96]	Ajisaka,T.. The Software Quark Model: A Universal Model for CASE Repositories. In Journal of Information and Software Technology, 1996.
[Brinkkemper 94]	Brinkkemper, S., Method Engineering: Engineering of Information Systems Development Methods and Tools. In Journal of Information and Software Technology, 1996.
[Coleman 92]	Coleman,F., Hayes,F. and Bear,S., Introducing Objectcharts or How to Use Statecharts on Object-Oriented Design. IEEE Trans Soft. Eng., Vol.18, No.1, pp.9 -- 18, 1992.
[De Marco 78]	DeMarco, T., Structured Analysis and System Specification, Yourdon Press, 1978.
[Harel 90]	Harel,D., Lachover,H., Naamad,A., Pnueli,A., Politi,M., Sherman,R. Shutull-Trauring,A. and Trakhtenbrot,M., STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Trans. Soft. Eng., Vol.16, pp.403 -- 414, 1990.
[Harmsen 94]	Harmsen, F., S. Brinkkemper, H. Oei, Situational Method Engineering for Information System Projects. In: Olle, T.W., and A.A. Verrijn Stuart (Eds.), Methods and Associated Tools for the Information Systems Life Cycle, Proceedings of the IFIP WG8.1 Working Conference CRIS' 94, North-Holland, pp. 169-194, Amsterdam, 1994.
[Harmsen 95]	Harmsen, F. and S. Brinkkemper, Design and Implementation of a Method Base Management System for a Situational CASE Environment. In: Proceedings of the APSEC' 95 Conference, IEEE Computer Society Press, Los Alamitos, CA, 1995.
[Harmsen 96]	Harmsen, F., and M. Saeki, Comparison of Four Method Engineering Languages. In: In: S. Brinkkemper, K. Lyytinen and R. Welke (Eds.), Method Engineering: Principles of Method Construction and Tool Support, Chapman & Hall, pp.209-231, 1996.
[Harmsen 97]	Harmsen, F., Situational Method Engineering. Moret Ernst & Young, 1997

[Hoef 95]	Hoef, R. van de, and F. Harmsen, Quality Requirements for Situational Methods. In: Grosz, G. (Ed.), In Proceedings of the Sixth Workshop on the Next Generation of CASE Tools, Jyväskylä, Finland, June 1995.
[Katayama 89]	Katayama, T., A Hierarchical and Functional Software Process Description and Its Enaction. In: Proceedings of 11th Int. Conf. on Software Engineering. pp.-343-352, May 1989.
[Klooster 97]	Klooster, M., S. Brinkkemper, F. Harmsen, and G. Wijers, Intranet Facilitated Knowledge Management: A Theory and Tool for Defining Situational Methods. In: A. Olive, J.A. Pastor (Eds.), Proceedings of CAiSE'97. Lecture Notes in Computer Science 1250, Springer Verlag, pp.303-317, 1997.
[Nuseibeh 95]	Nuseibeh, B., 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.
[Olle 91_J	Olle, T.W., J. Hagelstein, I.G. MacDonald, C. Rolland, H.G. Sol, F.J.M. van Asssche, A.A. Verrijn-Stuart, Information Systems Methodologies - A Framework for Understanding, 2nd Edition, Addison-Wesley, 1991.
[Rumbaugh 91]	Rumbaugh, J., Object oriented modeling and design, Prentice-Hall, Englewood Cliffs, 1991.
[Saeki 94]	Saeki, M., and K. Wen-yin, Specifying Software Specification and Design Methods. In: G. Wijers, S. Brinkkemper, T. Wasserman (Eds.), Proceedings of CAiSE'94, Lecture Notes in Computer Science 811, Springer Verlag, pp. 353-366, Berlin, 1994.
[Slooten 96]	Slooten, K. van and B. Hodes, Characterizing IS Development Projects. In: S. Brinkkemper, K. Lyytinen and R. Welke (Eds.), Method Engineering: Principles of Method Construction and Tool Support, Chapman & Hall, pp.29-44, 1996
[Song 95]	Song, X., A Framework for Understanding the Integration of Design Methodologies. In: ACM SIGSOFT Software Engineering Notes, Vol. 20, No. 1, pp. 46-54, 1995.
[Sorenseon 88]	Sorenson,P.G., J.P.Tremblay, A.J.McAllister, The Metaview System for Many Specifications Environements. In IEEE Software, Vol.30, No.3, pp.30-38, 1988.
[Ward 85]	Ward,P., S. Mellor, Structured Development for Real-time Systems, Yourdon Press, 1985.