Embedding Metrics into Information Systems Development Methods: An Application of Method Engineering Technique

Motoshi Saeki

Dept. of Computer Science, Tokyo Institute of Technology
Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan
saeki@cs.titech.ac.jp

Abstract. Many methods for information systems development such as object oriented analysis and design (OOA & OOD) support the activities to construct artifacts, but do not include the activities to measure the quality of the artifacts or to improve them based on the results of their measurement. In this paper, by using a meta modeling technique, we propose a general framework of extending an existing method into the method which includes the activities for attaching semantic information to the artifacts and for measuring their quality using this information. Embedding this information into a method can be considered as a method assembly of the meta model of the method and the metrics model. We can formally represent this process with Method Engineering Language (MEL). We show an example that our technique is applied to; the extended version of a use case modeling method.

1 Introduction

Various methods for information systems development such as object oriented methods[10] have been proposed in this decade and are being put into practice. These methods only support the human activities to construct artifacts, but do not include the activities to measure the quality of the artifacts and/or to improve them based on the results of their measurement. Since there are wide varieties of artifacts for methods, we should have different metrics to measure the quality for the methods and it is necessary to define the metrics according to the methods. For example, in the method where we develop a class diagram, we can use the CK metrics[8] and improve artifacts based on the values of CK metrics. The technique of Cyclomatic number[13] can be applied to the construction of an activity diagram of UML (Unified Modeling Language)[15] in order to measure its complexity, so that we can avoid constructing complicated activity diagrams. These examples show that effective metrics vary on a method.

The existing metrics such as CK metrics and Cyclomatic number are for expressing the structural and syntactical characteristics of artifacts only, but do not reflect their semantic aspects. Suppose two class diagrams of Elevator Control System, which are the same except for the name of a class; one includes the class named "A" and in the other one the class is named "Elevator". Although these diagrams are completely the same in graph structural view, the latter diagram has higher quality rather than the former, because the latter uses the domain-specific word "Elevator" and every reader can easily understand what this class denotes. However, the existing metrics provides the same value for these two diagrams. This example shows that the metrics expressing semantic aspects allows us to measure the quality of artifacts more correctly and precisely. It is difficult to extract automatically semantic information from the complete artifacts, thus a developer can and should attach this semantic information to the artifacts (including intermediate ones) by manual during his development activities. That is to say, the method that the developer adopts should include the supports of attaching semantic information. And the method should also suggest how to calculate the measures from the attached semantic information. In the above example of the class diagrams of "Elevator", the example method should support the human activity of measuring the occurrences of domain specific words in a class diagram and suggest how to improve the quality of the diagram based on this measurement. The measure of the occurrences of domain specific words is specific to this method. To embed the function of measuring artifact quality into methods, we should explore the techniques how to build the methods supporting the metrics of artifacts, including the definition of the metrics.

Method engineering is a discipline for exploring techniques to build project-specific methods[4], and meta modeling techniques to specify methods semi-formally or formally have been developed to manipulate the methods by computer. It provides the technique called method assembly to compose a new project-specific method from a set of meaningful parts of the existing methods[5, 14,17]. Meaningful parts of methods are called method fragments or method chunks. A method assembly technique can be used to embed metrics models into the existing methods and to get a new method that has the function to measure the quality of artifacts. In the above example of "Elevator" class diagram, we have two method fragments; 1) a meta model of class diagram method as a base, and 2) a metric model based on the occurrences of domain-specific words in class diagrams. When we assemble these two method fragments, we can get a new method that can measure the quality of class diagrams based on the occurrences of domain-specific words. Thus a meta modeling technique is used to define metrics model according to a method, and the assembly technique provides a general framework how to tailor an existing method into a new method that can measure the quality of artifacts.

In this paper, by using a meta modeling technique and a method assembly one, we propose a general framework of extending an existing method into the method which includes the activities for attaching semantic information to the artifacts and for measuring their quality. Semantic information and the definition of the measures are formally defined on a meta model, and they are embedded into a meta model of an original method by performing a method assembly. We use the formal language Method Engineering Language (shortened to MEL), since it provides a unified framework to describe both of method fragments and method assembly processes. Furthermore, we present an example that our technique is applied to; the extended version of a use case modeling method. In the example, we get the use case modeling method where the quality of a use case diagram can be calculated. It can be formalized as method assembly processes and be defined in MEL. In this sense, this paper shows the expressive power of MEL through the application to the example.

In the next section, we introduce how to describe method fragments, i.e. a meta modeling technique. Section 3 presents a general framework how to extend methods on its meta model. We call this extended version of a method a measurable method. Section 4 presents the example of defining the metrics for use case diagrams, and we show how to get the measurable method by using method assembly.

2 Meta Modeling Technique and MEL

Although the textbooks and manuals of a development method such as object oriented methods contain narrative texts written in natural language, figures and examples for ease of understanding, there are several techniques to model a method formally or semi-formally in order to manipulate the method by computer, e.g. to generate CASE tools based on it[12]. These techniques are called meta modeling techniques. Roughly speaking, methods consist of two facets; one is artifact and the other is activity. The artifact facet specifies what artifacts are developed following a method, e.g. class diagrams, while the activity one specifies the processes to develop the artifacts, e.g. “Identify objects and classes at first” in object oriented methods. Figure 1 shows a simplified example of a meta model for the method of constructing a use case diagram of UML diagrams[15]. In the figure, the structure of the artifact is defined in a class diagram, while the activities and their execution order are specified in an activity diagram. Dotted arrows (object flows) stand for the artifacts that are produced by an activity. For example, the second activity “Identify Use Cases” produces a set of the identified “Use Cases” and their relationships to “Actors”.

Although we use a class diagram and an activity one of UML on account of the easiness to understand, we describe the meta models in more formal style by using Method Engineering Language (MEL)[6]. Figure 2 shows the MEL descriptions of Figure 1. In MEL, method fragments are classified into two types ; product fragment and process one. The description of a product fragment in MEL can be considered as a textual representation of a class diagram which defines the structure of the products shown in Figure 1 (a). The description that begins with the reserved word “PRODUCT” defines the concept of product elements included in a method and corresponds to a class of Figure 1(a). On the other hand, the word “ASSOCIATION” declares the relationship between concepts, i.e. an association between the classes. The description of a process fragment corresponds to a textual representation of an activity diagram. MEL has syntactic constructs for composing more complicated activities from the

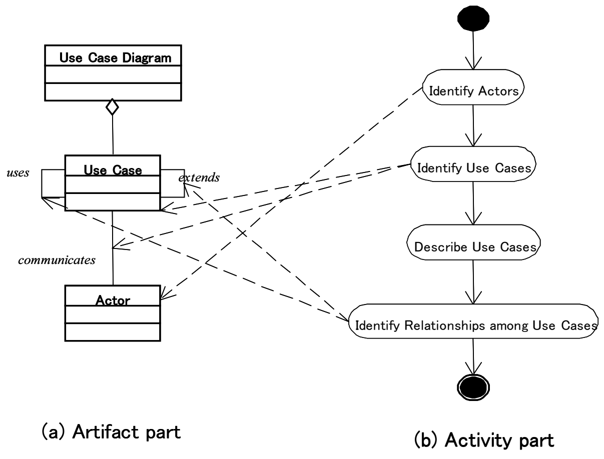


Fig. 1. An Example of A Meta Model

activities, e.g. sequential execution, conditional branch, iteration and parallel execution including fork and join, etc. Readers can find the activity "Construct a use case diagram" is composed as a sequential execution of the four activities, as shown in Figure 2(b). The reserved word "REQUIRED" specifies the input products for the process.

```
PRODUCT Use Case Model;
  ID Use Case Model;
  IS_A Diagram, Structured Text;
  LAYER Diagram;
  PART OF Analysis Model;
  NAME TEXT;

PRODUCT Use Case:
  LAYER Concept;
  PART OF Use Case Model;
  SYMBOL Oval;
  NAME TEXT;
  ASSOCIATED WITH {(uses, ), (extends,),
     (hasUseCase,), (communicates,)}.

ASSOCIATION hasUseCase:
  ASSOCIATES (Use Case Diagram, Use Case);
  CARDINALITY (1..n; 1..1).
```

(a) Product Fragment

```
PROCESS Construct a Use Case Diagram:
  LAYER Diagram;
  TYPE Creation;
  PART OF Create an Analysis Model;
  REQUIRED {Interview results};
  REQUIRED OPTIONAL
       Current Information system;
  (- Identify Actors ;
   - Identify Use Cases ;
   - Describe Use Cases ;
   - Identify Relationships among Use Cases ;
  )
  DELIVERABLES {Use Case Diagram}.
```

(b) Process Fragment

Fig. 2. An Example of MEL Descriptions

3 Method Assembly for Measurable Methods

Almost of all existing metrics for measuring the quality of an artifact is based on its syntactical structure. For example, in CK metrics, the number of operations[1] per class (WMC), the depth of an a class from a root in an inheritance tree (DIT) and so on are calculated, in order to express the complexity of an object oriented design from various viewpoints. These measures do not reflect semantic aspects of the artifacts. Suppose that many operations having the same function occur in the different classes in an object oriented design. This design cannot be said to be a good design, because we should modify all of the occurrences of the operations if we change the function, i.e. it has low modifiability. To quantify this situation, we need the semantic information on which operations have the same function, and this type of information cannot be automatically obtained from the syntactic characteristics of the design document. A developer should add this information by manual during his development activities. More concretely, he investigates which operations have the same function and attach this information as attributes of the artifacts. And he also defines how to calculate the metrics from the attached attributes. It greatly depends on development methods what attributes to attach and how to quantify quality characteristics.

Furthermore McCall et. al. proposed that a quality characteristic such as completeness could be quantified by a weighted arithmetic average of the metrics [7], and these weight factors also depend on the methods. In this sense, we extend the existing method into a new one whose meta model has 1) attributes in its artifact part, 2) activities for investigating the attribute values of the artifact, 3) expressions for calculating metrics from the attribute values, and 4) mathematical expressions such as weighted arithmetic average for quantifying quality characteristics based on the metrics. Figure 3 shows a general framework for extending an existing method on its meta model. The upper left part Meta Model (Method Description) of the figure shows a general description in an existing method, where the activity "Identify Artifact Elements" produces the elements of the artifacts. On the other hand, in the lower left part Metrics Model Definition, we define attributes, metrics, weighted factors. It also includes the activity of attaching the semantic information of artifacts as attribute values ("Describing Attribute Values") and the activity of using quality characteristics for artifact improvement ("Considering Quality Characteristics"). The classes "Metrics" and "Quality Characteristics" specify the measures using "Attribute" and the expressions for calculating quality characteristics from the metrics respectively. By assembling these two parts to a new method, we can get the right part of the figure. We call it a measurable method of the existing method. This assembly process can be formally specified with specific manipulation operators of MEL on the meta models as mentioned in [6,9].

[1] We should have used the term "methods" instead of "operation". To avoid the confusion to a development method, however we use the word "operation" in this paper.

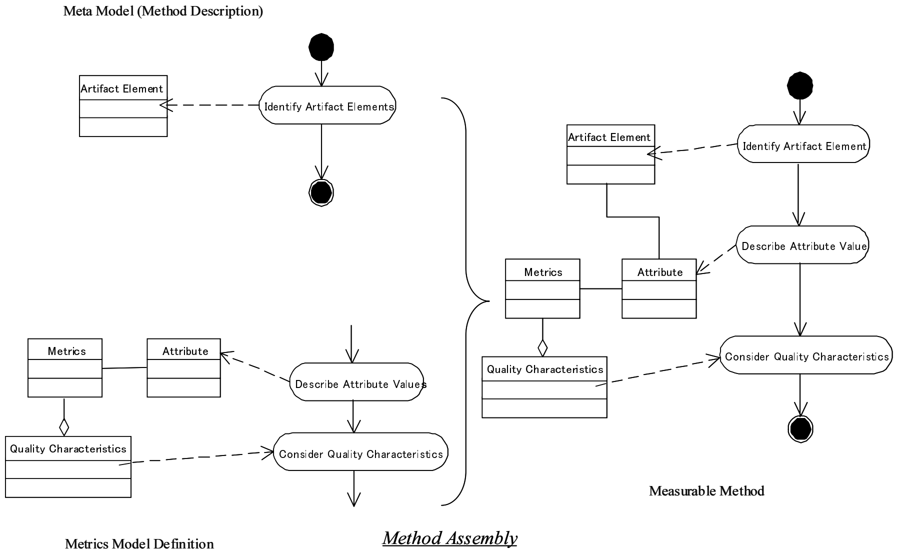


Fig. 3. Method Assembly for Measurable Methods

The above is a very simple but general framework for producing measurable methods. The point is that we can produce measurable methods by means of applying method assembly to the existing, non-measurable methods. Let's consider another but also general example to get measurable methods by using method assembly.

GQM (Goal-Question-Metric) model has been proposed in [3] as a framework for defining and interpreting software measurement. Its model includes three basic concepts; 1) Goal: the purposes of the measurement, in more detail, a goal specifies what objects are measured for what purposes from which viewpoints with respect to which focuses, 2) Question: Questions are for clarifying the attainment of the Goal and for assessing the Goal, and 3) Metric: a set of data to quantitatively answer the corresponding Questions. A class diagram of the upper left part GQM Meta Model in Figure 4 illustrates a meta model of GQM products, while an activity diagram in the part shows the process to enact GQM framework for measurement. At first, developers define a GQM model, i.e. define Goals, generate Questions for each Goal and define Metric for the Goals, and then collect the data according the specification of the Metric. After that, they interpret the data to evaluate the Questions and the Goals.

The application of GQM framework to measure the quality of artifacts can also be considered as the method assembly of a GQM and non-measurable methods, as shown in Figure 4. Metric is connected to artifact elements through its "Attribute" values, and the activities of GQM are embedded in the form of parallel composition to the activities of the non-measurable method. In [11], GQM approach was applied to meta data management in data warehouses, and the GQM meta model was adapted to schemas of data warehouses. In this sense, it can be considered as a method adaptation in an artifact aspect only. On the other hand, our technique includes method adaptation and assembly not only for artifact parts but also activity parts.

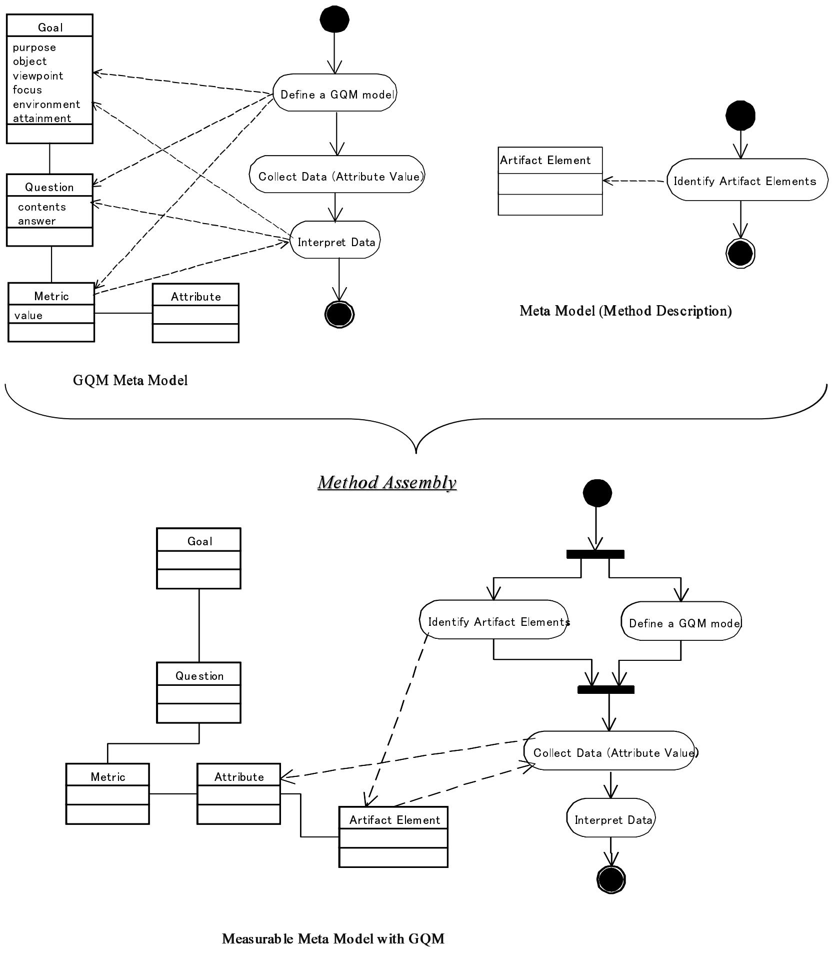


Fig. 4. GQM Meta Model and Its Method Assembly

4 Defining Metrics with MEL: An Example of Measurable Use Case Modeling Method

4.1 Modifiability on Use Case Diagrams

Quality characteristics for requirements specifications are listed up in IEEE830 standards[1], e.g. completeness, correctness, modifiability etc. One of the typical metrics on a use case diagram is the number of the use cases appearing in the diagram, and it is based on syntactical features of the diagram. Since it is insufficient to quantify the quality characteristics of IEEE 830, we should consider the metrics relevant to the meaning of use cases. For simplicity, we focus on the metrics relevant to modifiability. If we modify a use case, its modification can propagate to the use cases that are connected to it with "<<extends>>" or "<<uses>>" relationships. The number of the occurrences of these relationships among use cases can be a measure denoting modifiability. This measure can be calculated directly from the structural characteristics of a use case diagram. Let's consider other measures that are more relevant to the meaning of use cases. Based on [16], we adopt two additional dependency relationships among the use cases; data dependency and control dependency, and the concept of basic types of use cases called use case types. An analyst identifies both of them during his activities for constructing a use case diagram, and it means that he should perform additional activities, e.g. "Identify Control Dependencies", "Identify Data Dependencies" and "Identify Use Case Types" after finishing the activity "Identify Use Cases".

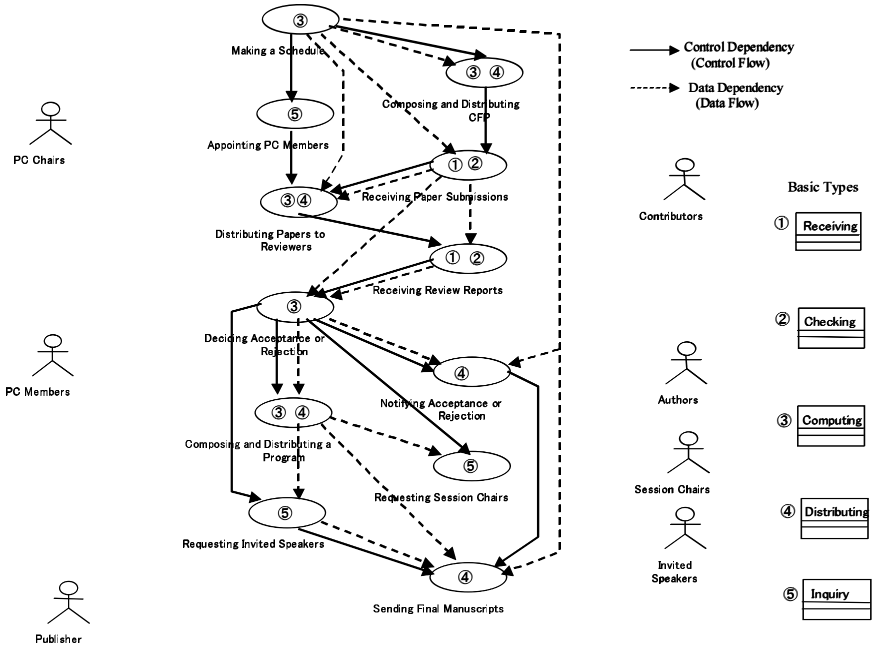

Making a Schedule
Composing and Distributing CFP
Appointing PC Members
Receiving Paper Submissions
Distributing Papers to Reviewers
Receiving Review Reports
Deciding Acceptance or Rejection
Notifying Acceptance or Rejection
Composing and Distributing a Program
Requesting Session Chairs
Requesting Invited Speakers
Sending Final Manuscripts
PC Chairs
PC Members
Publisher
Contributors
Authors
Session Chairs
Invited Speakers
Control Dependency (Control Flow)
Data Dependency (Data Flow)
Basic Types
① Receiving
② Checking
③ Computing
④ Distributing
⑤ Inquiry

Fig. 5. An Example of Use Case Diagrams

Figure 5 illustrates a use case diagram together with dependency relationships and the information on use case types. For simplicity, we omit the interactions between the actors and the use cases in the figure. This example is a tool for supporting tasks that program committee chairs (PC chairs) of academic international conferences have to perform. The PC chairs should compose the CALL FOR PAPER (CFP) and organize the committee. They receive paper submissions and distribute them to the reviewers, normally PC members. After getting the review reports from the reviewers and summarizing them ("Receiving review reports" in the figure), they have PC meetings to decide which papers will be accepted or rejected. The authors of the papers are notified of their acceptance or rejection by the PC chairs.

A control dependency expresses the execution order on use cases. For example, after the use case "Notifying acceptance or rejection" is performed, "Sending final manuscripts" should be done. A data dependency relationship represents which use cases consume the data that another use case produces, and is also significant to identify the use cases that we should modify. The modifications on a use case can propagate the different use cases that have control or data dependencies to it, so these occurrences has an influence on the difficulty in the modifications. The more they occur in the diagram, the more difficult its modification is.

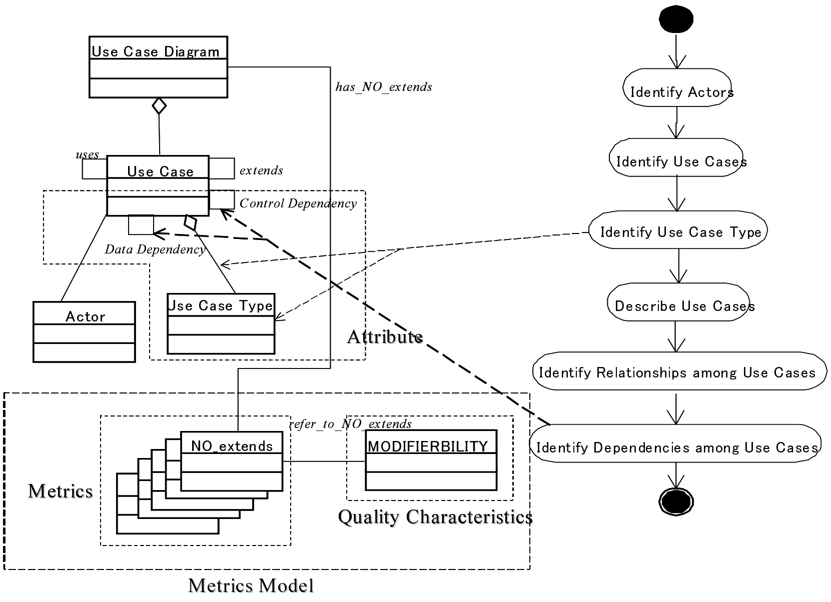


Fig. 6. A Meta Model of Measurable Use Case Modeling Method

A use case type is a kind of an abstraction or a semantic category, which stands for abstract meaning of a use case, and a set of the use case types depends on an application domain. In the domain of the example like a data processing system, we use five use case types "Receiving", "Checking", "Computing", "Distributing" and "Inquiry", all of which are the manipulation on data. Considering the purpose of executing a use case, an analyst attaches these types to the identified use cases. The circled numbers in a use case in the figure shows the use case types that are attached to it. For example, the use case "Receiving Paper Submission" has the purposes of receiving paper submissions as data and of checking if the submitted papers meet a certain format or not. That is to say, the two types "Receiving" and "Checking" are attached to it. In this case, it has the two roles, i.e. compounded meaning and its modifications can be complicated because we should consider both of these two roles simultaneously during the modification process. Thus we can use the number of attached use case types to each use case as a measure of modifiability.

Figure 6 depicts a meta model, and it is the result of embedding these attributes for quantifying modifiability to the existing meta model, according to the general framework in Figure 3. For example, readers can find that the concept "Use Case Type", the associations "Control Dependency" and "Data Dependency" are added as attributes to the product part of the meta model (product fragment). As for a process part of the meta model (process fragment), two activities for the metrics are added. For example, the activity "Identify Use Cases Types" is for finding the types of the identified use cases. The activity for control and data dependencies "Identify Dependencies among Use Cases" is also added in the process part. We call this extended version of use case modeling method a measurable use case method.

As mentioned above, we have two types of metrics from which the modifiability can be indirectly measured; one is NOD (Number of Dependencies) for denoting the ratio of how many relationships occur among use cases and the other one NUCT (Number of Use Case Types) is the multiplicity ratio of multiplicity of use case types attached to use cases. The definition of the metrics can be defined by using the meta model as follows;

$$\mathrm{NOD} = \frac{AllDependencies - \#Dependency}{AllDependencies}$$
$$\text{where } AllDependencies = (\#UseCase \times (\#UseCase - 1))/2.$$

$$\mathrm{NUCT} = \frac{1}{AVE_{u \in UseCase}\{\#\{ut \in UseCaseType \mid aggregates(u, ut)\}\}}$$

Note that *AllDependencies* denotes the number of all possible dependencies on a use case diagram from graph theoretic view, i.e. combinations between arbitrary two use cases. We should explain conventional notation appearing in the above expressions. The name of each class and the name of each association denote a set and a predicate respectively. For instance, $UseCase$ and $u \in UseCase$ denote a set of the identified use cases and a use case u respectively. We use "*aggregates*" as the predicate name of an object aggregation relationship. For instance, $aggregates(u, ut)$ denotes that the use case u has a use case type ut. $\#S$ stands for the number of the elements of the set S. $AVE_{p(x)}\{s(x)\}$ means the average value of a set of the numbers $s(x)$ constructed from x such that $p(x)$. *Dependency* appearing in NOD is either *extends*, *uses*, *Control Dependency* or

Data Dependency. Thus we have four variations of the metrics related to a kind of dependency among use cases, and we call them NO_extends, NO_uses, NO_CD and NO_DD in order. In the example, these values are 1, 1, $(66 - 13)/66 = 0.8$ and $(66 - 15)/66 = 0.77$ because $AllDependencies = (12 \times (12 - 1))/2 = 66$. Since NUCT is the reciprocal number of an average of attached use case types for each use case, we can get NUCT$= 12/17 = 0.71$.

Finally, as McCall did, we adopt weighted arithmetic average to quantify modifiability from these five metrics values and we can get it as follows;

$$MODIFIABLITY =$$
$$w_1 \times \text{NO\_extends} + w_2 \times \text{NO\_uses} + w_3 \times \text{NO\_CD} + w_4 \times \text{NO\_DD} + w_5 \times \text{NUCT}$$
$$\text{where } w_1 + w_2 + w_3 + w_4 + w_5 = 1 \text{ and } 0 \leq w_i \leq 1 (i = 1, ...5).$$

This expression and the weighting factors are specified in an instance of class "Quality Characteristics" as shown in Figure 3, and much experience can determine their values. In this paper, taking the same value, i.e. 0.2 for each, as an example, we get the value $0.2 \times 1 + 0.2 \times 1 + 0.2 \times 0.8 + 0.2 \times 0.77 + 0.2 \times 0.71 = 0.2 + 0.2 + 0.16 + 0.15 + 0.14 = 0.85$ as MODIFIABILITY quality. By using this technique, we can quantify the other quality characteristics of a use case diagram as a requirement specification.

```
PROCESS Extend Product Fragments
          to Measurable Use Case Modeling Method ;
  TYPE ASSEMBLY
  REQUIRED {Use Case Model, Use Case Type,
    Control dependency, Data dependency, Metrics Model};
  (
  Join Use Case Model
   With Control dependency,
      Data dependency, Use Case Type,
      Metrics Model
  Through
      has_NO_extends(Use Case Diagram, NO_extends)
      has_NO_uses(Use Case Diagram, NO_uses)
      has_NO_CD(Use Case Diagram, NO_CD)
      has_NO_DD(Use Case Diagram, NO_DD)
      has_NUCT(Use Case Diagram, NUCT)
      has_Use_Case_Type(Use Case, Use Case Type)
  Into Attributed Use Case Model
  )

PRODUCT Use Case Type:
 LAYER Concept;
 NAME [Receiving, Checking, Computing, Distributing, Inquiry]

ASSOCIATION Control dependency:
 ASSOCIATES (Use Case, Use Case);
 CARDINALITY (0..n; 0..n).
```

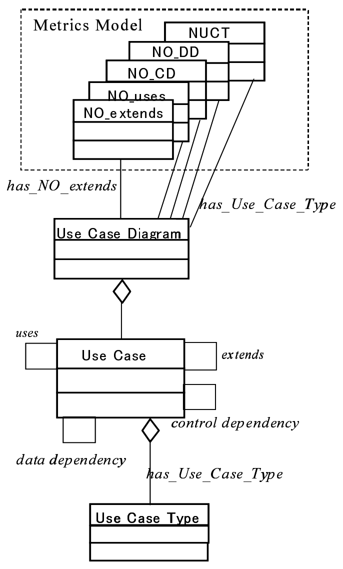


Fig. 7. Method Assembly for Product Fragments of Measurable Use Case Modeling Method

Figure 6 shows a meta model of the measurable use case modeling method that can quantify the modifiability of a produced use case diagram, by using above technique. Each class whose category is "Metrics" in the figure denotes a metrics such as No_extends and No_uses. The class "MODIFIABILITY" defines how to calculate the modifiability from the metrics. The details how to calculate quality characteristics will be mentioned in the next subsection.

```
PROCESS Extend Process Fragments
        to Measurable Use Case Modeling Method;
  TYPE ASSEMBLY
  REQUIRED {Construct a Use Case Diagram,
        Identify Use Case Type,
        Identify Dependencies among Use Cases};
  (
  Let Identify Use Case Type
   Precede Describe Use Cases
  Let Identify Relationships among Use Cases
   Precede
      Identify Dependencies among Use Cases
  )
  DELIVERABLES
   {Construct Measurable Use Case Model}.
```

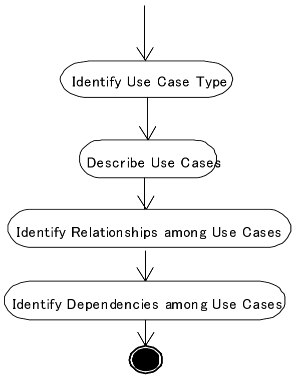

Identify Use Case Type
Describe Use Cases
Identify Relationships among Use Cases
Identify Dependencies among Use Cases

Fig. 8. Method Assembly for Process Fragments of Measurable Use Case Modeling Method

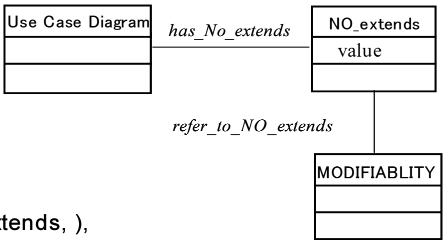
4.2 Defining Measurable Methods by Method Assembly

As shown in Figure 6, the measurable use case modeling method is an extension of a use case modeling method and we can get it by assembling some product fragments and process ones. In MEL, we can define a method assembly process as a set of the process fragments that specify how to add new fragments. It consists of two fragments; one is for assembling product fragments and another is for process ones.

Figure 7 illustrates a part of a method assembly process of Use Case Model and Metrics Model, in order to get a product fragment of the measurable use case modeling method. The readers can find that a concept as a class "Use Case Type" and two associations, e.g. Control Dependency and Data Dependency are newly added to the original method, by using "Join ... With ... Through ... Into" statement of MEL. "Join F1 with F2 Through A1 into F3" is an operation to assemble a product fragment F1 and a fragment set F2 by using associations A1, and the fragment F3 is a result of the operation. In this figure, additional six associations, e.g. "*has_NO_extends*" and "*has_Use_Case_Type*" are adopted to connect Use Case Type and Metrics Model to Use Case Model. A part of the assembled fragments like Metrics Model will be shown later in Figure 9, and Metrics Model is a meta model which consists of Metrics concepts (NO_extends, NO_uses, NO_CD, NO_DD, NUCT) and Quality Characteristics concept (Modifiability).

Figure 8 shows an assembly process for process fragments of the measurable use case modeling method. Two new process fragments as activities "Identify Use Case Types" and "Identify Dependencies among Use Cases" are added to the original process fragment of use case modeling method. The assembly operation "Let P1 Precede P2" creates an execution order between the process fragments P1 and P2. Note that REQUIRED section specifies which process fragments are assembled as inputs of the assembly process "Extend Process Fragments to Measurable Use Case Modeling Method".

```
PRODUCT NO_extends;
 LAYER Concept;
 PART OF Metrics Model;
 NAME TEXT;
 ATTRIBUTES
     value : REAL
 ASSOCIATED WITH {(has_No_extends, ),
            (refer_to_No_extends, )}.
 (RULE :
   value
     = (AllDependencies - #Instances(extends)) / AllDependencies ;
   AllDependecies = (#Instances(Use Case) * (#Instances(Use Case)-1)/2;
 ).
```



```
PRODUCT MODIFIABILITY:
 LAYER Concept;
 PART OF Metrics Model;
 NAME TEXT;
 ATTRIBUTES
     value : REAL
 ASSOCIATED WITH
   {(refer_to_No_extends,),
   (refer_to_No_uses,), (refer_to_CD,),
   (refer_to_No_DD,), (refer_to_NUCT,)}.
 (RULE :
   value =
     w1*refer_to_NO_extends.value
     + w2*refer_to_NO_uses.value
     + w3*refer_to_NO_CD.value + w4*refer_to_NO_DD.value
     + w5*NUCT
 ).
```

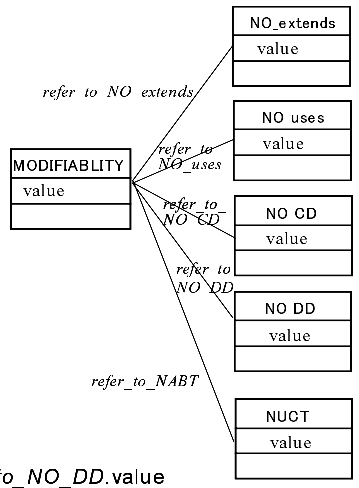


Fig. 9. Definitions of Metrics and Quality Characteristics with MEL

Finally, we need to define product fragments which have the functions to calculate quality characteristics like MODIFIABILITY. Figure 9 illustrates the definitions of the metrics NO_extends and the quality characteristics MODIFIABILITY. These definitions include their calculating expressions and weighting factors in "rule" section of MEL. The rule section also specifies invariants which hold on attribute values. For example, the attribute "value" in NO_extends is calculated by means of an equation of the rule section.

Figure 10 shows how to calculate the modifiability of a use case diagram according to the definitions of Figs 6, 7, 8 and 9. It is written in UML object diagram, because it can be considered as an instance of the meta model of Figure 6.

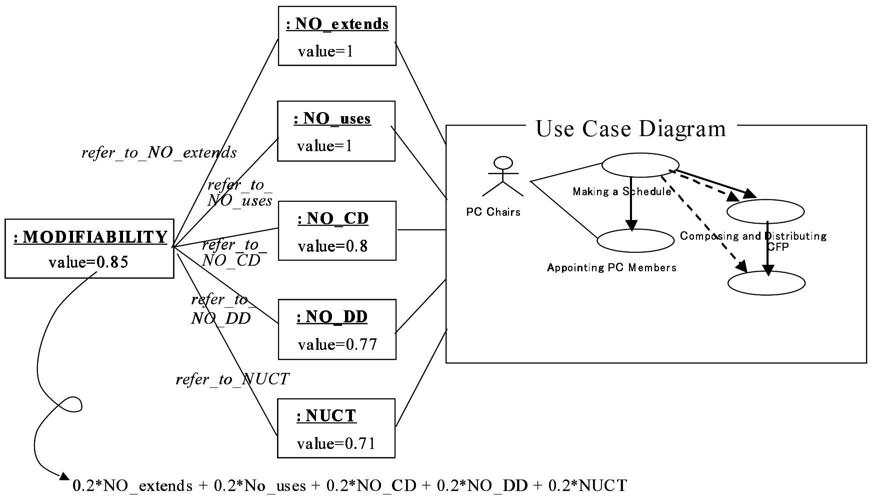


Fig. 10. Calculating Quality Characteristics

5 Conclusion and Research Agenda

This paper presented a general framework how to embed quantification techniques to the existing development methods by using method engineering techniques. The meta models of a measurable method allows us to generate (semi-) automatically CASE tools supporting measurement activities and the improvement of product quality, by means of a CAME (Computer-Aided Method Engineering) tool such as Decamerone[9], Mentor[18] and MetaEdit+[12].

Research agenda for the future directions can be summarized as follows;

1. Refining a technique for modeling and defining metrics. In this paper, we use usual mathematical expressions to define the metrics. Like [2], the usage of Object Constraint Language (OCL)[20] is one of the promising approaches to express wide varieties of metrics easily and to interchange their definitions among different development organizations.

2. Collecting metrics definitions. To assess our technique, we need various kinds of metrics and apply our technique to them. We are exploring the metrics for quantifying different quality characteristics on wide varieties of artifacts and are trying to formalize both of metrics and methods.
3. Usage of metrics. Since our technique includes defining development methods, we should consider how to use the defined metrics in order to improve the quality of artifacts during development processes. More concretely, we should clarify what activities improve the artifacts by using the metrics in the method, like GQM[3].
4. Support of activities for identifying semantic information (attributes). To get the precise quality characteristics, a developer identifies semantic information of product elements. In the example of the measurable use case modeling method, a developer identifies use case types and it is a problem how to set up a set of use case types in advance according to problem domain. This type of semantic information is closely related to domain ontology[19] and the ontology is helpful to develop metrics to measure artifact quality in a semantic level. MEL has another level of an ontology system for method concepts[9]. The usage of method ontology included in MEL is also a promising approach.

Acknowledgements. The author would like to thank Sjaak Brinkkemper and Frank Harmsen for their insightful suggestions to Method Engineering Language.

References

1. IEEE Recommended Practice for Software Requirements Specifications. Technical report, IEEE Std. 830-1998, 1998.
2. F. B. Abreu. Using OCL to Formalize Object Oriented Metrics Definitions. In *Tutorial in 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2001)*, 2001.
3. V. Basili. Software Modeling and Measurement: The Goal/Question/Metric Paradigm. Technical Report UMIACS-TR-92-96, University of Maryland, 1992.
4. S. Brinkkemper. Method Engineering: Engineering of Information Systems Development Methods and Tools. *Information and Software Technology*, 37(11), 1995.
5. S. Brinkkemper, M. Saeki, and F. Harmsen. Meta-Modelling Based Assembly Techniques for Situational Method Engineering. *Information Systems*, 24(3):209 –228, 1999.
6. S. Brinkkemper, M. Saeki, and F. Harmsen. A Method Engineering Language for the Description of Systems Development Methods (Extended Abstract). In *Lecture Notes in Computer Science (Proc. of CAiSE'2001)*, volume 2068, pages 473–476, 2001.
7. J.P. Cavano and J.A. McCall. A Framework for the Measurement of Software Quality. In *Proc. of ACM Software Quality Assurance Workshop*, pages 133–139, 1978.
8. S. Chidamber and C. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Trans. on Software Engineering*, 20(6):476–492, 1994.
9. F. Harmsen. *Situational Method Engineering.* Moret Ernst & Young Management Consultants, 1997.

10. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
11. M. Jarke, M. Jeusfeld, C. Quix, and P. Vassiliadis. Architecture and Quality in Data Warehouses. In *Lecture Notes in Computer Science (CAiSE'98)*, pages 93–113. Springer-Verlag, 1998.
12. S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+ : A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Lecture Notes in Computer Science (CAiSE'96)*, volume 1080, pages 1–21, 1996.
13. T. McCabe and C. Butler. Design Complexity Measurement and Testing. *CACM*, 32(12):1415–1425, 1989.
14. J. Ralyte and C. Rolland. An Assembly Process Model for Method Engineering. In *Lecture Notes in Computer Science (Proc. of CAiSE'2001)*, volume 2068, pages 267–283, 2001.
15. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
16. M. Saeki. Reusing Use Case Descriptions for Requirements Specification : Towards Use Case Patterns. In *Proc. of 6th Aisa-Pacific Softwrae Engineering Conference (APSEC'99)*, pages 309–316, 1999.
17. M. Saeki. Role of Model Transformation in Method Engineering. In *Lecture Notes in Computer Science (Proc. of CAiSE'2002)*, volume 2348, pages 626–642, 2002.
18. S. Si-Said, Rolland C., and G. Grosz. MENTOR : A Computer Aided Requirements Engineering Environment. In *Lecture Notes in Comupter Science (CAiSE'96)*, volume 1080, pages 22–43, 1996.
19. Y. Wand. Ontology as a Foundation for Meta-Modelling and Method Engineering. *Information and Software Technology*, 38(4):281–288, 1996.
20. J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison Wesley, 1999.