Requirements Eng (2006) 11: 221–237
DOI 10.1007/s00766-005-0026-1

ORIGINAL RESEARCH

Naveen Prakash

On generic method models

Received: 9 August 2005 / Accepted: 9 December 2005 / Published online: 10 March 2006

Abstract The generic method model assumes that methods abound and method engineering and application engineering is done in diverse areas. Therefore, it is necessary to understand the notion of a method independent of information systems and software engineering. The generic model presented here abstracts out from product and process meta-models to define a system of concepts that can be used in any domain. Additionally, it integrates in it essential features of methods like quality checking, guidance, backtracking, and traceability. The proposed generic model looks upon a method in two parts, the static and the dynamic parts. The former provides the basic structure of a method whereas the latter is the enactment support provided for application development. The static part provides the notion of method blocks and dependencies between them. Method blocks can be enacted. The generic model is a triple <M, D, E> where M is the set of method blocks, D is the set of dependencies between method blocks, and E is the enactment mechanism.

1 Introduction

A method has been described in various ways. The Oxford dictionary [1] considers it to be (a) 'a particular way of doing' and (b) 'the quality of being well planned and organized'. Both these attitudes have been adopted in Software Engineering and Information Systems. The *first attitude* is reflected in approaches [2–5] that are based on the way of working adopted by the engineer to carry out a task. The *second attitude* results in the view that a method provides the means for well planned and organized product development. Thus, [6] looks upon a method as providing the necessary structure using which a product is built. A method is a collection of tools and techniques, product and process models, guidelines, checklists, heuristics, etc. that help an application engineer to build a suitable product.

For information systems development methods (ISDMs), the basic technique adopted to understand and engineer methods is that of *meta-modelling*. Meta-models have been developed with different abstractions and for highlighting different aspects of methods, the product aspect [7, 8], the process aspect [9, 10], and integrated product–process aspect [11, 12]. Considering the crucial importance of guidance and tracing capabilities in methods, a system of three meta-models was defined in [13]. These are (a) 'basic' meta-model, (b) guidance meta-model and (c) traceability meta-model. However, quality issues, those of representing method constraints and heuristics as well as of determining constraint and heuristic satisfaction have remained unaddressed in meta-models.

A formal definition of a method [9, 10] was made in the early 1990s. Here a method was treated as an *n*-tuple with *n* being around 10. Later, with the emergence of situational method engineering, the notion of a method fragment was formalized [14]. This formalization uses ten sets and five predicates resulting in a fragment based method being treated as a 15-tuple. Evidently, the number of concepts used to represent a method is very large and a method is a very complex artefact.

In the last few years we have seen the emergence of *generic models* that abstract out the common properties of meta-models. This results in a three-layer framework consisting of the generic, meta-model and method layers [12, 15]. This idea was tested in [16] and the benefits of the generic layer were found to be as follows:

- The generic layer makes explicit nature of the domain. Therefore, it promotes the development of methods and processes specific to that domain. This improves the chances of method acceptability.
- Again, by making explicit the nature of domain, it helps in understanding and comparing different domains.

N. Prakash
JUIT, 173215 Waknaghat, Himachal Pradesh, India
E-mail: praknav@hotmail.com

- The generic layer helps in meta-method engineering. It was shown that meta-models of very diverse domains could be engineered. Meta-models can be checked out for consistency and completeness under the generic view. This can be facilitated by constructing generic computer aided method engineering (generic CAME) tools.
- The generic layer can be used for simulating real processes and systems before they are actually implemented. For example, in the robotics domain, the entire robot can be conceptualized, and a method produced. Using computer aided systems engineering (CASE) support, application processes can be built to check that the robot does, in fact, handle the proposed range of tasks. If not then the robot can be re-conceptualized. Thus, before constructing the real robot, its viability can be fully established.

Given this range of benefits of the generic layer, our focus now is on the development of a generic model. Our aim is to build a model that is

1. Simple. Simplicity is measured in terms of the arity of the tuple required to formalize it. Thus, we are looking for much less than the 10 to 15-tuple formalizations referred to above.
2. Complete. Completeness is to be seen in the range of method features, guidelines, heuristics, etc. that can be derived from the model.

The generic model developed here treats a method as a triple, < M, DEP, E) where M is a set of method blocks, DEP is a set of dependencies between method blocks, and E is an enactment mechanism for method block enactment. Clearly, this three tuple is relatively simpler than the formalizations referred to above. Additionally, instead of treating guidance, traceability and basic method capabilities separately as in [13], we put them together in a unified whole. The set, DEP, forms the basis of this unification. Finally, our generic model addresses quality issues as well. This happens in three parts. First, constraints and heuristics are represented in a sub-class of method blocks called quality checking method blocks. Second, dependencies tell us which quality method blocks are to be enacted when and finally, quality method block enactment under the enactment mechanism provides information on satisfaction. With this capability, the proposed generic model provides a relatively more complete representation of method features.

Our approach to generic model definition starts off, in the next section, with building a classification framework that helps us to define our notion of genericity. Thereafter, in Sect. 3, we identify the properties that generic models should have. The implication of these properties on generic model development is considered in Sect. 4. The generic model is presented in the Sect. 5. This section is divided into two main parts, the first dealing with the static part and the second with the dynamic part. The underlying mechanisms for backtracking, traceability, and guidance are explained here. The section ends with our Enactment Mechanism. Appendix 1 contains an example of use of the generic model and Appendix 2 contains a glossary of terms.

2 The four-dimensional classification framework

Methods have been classified in different ways.

1. Methods are classified on the basis of *their emphasis*. Those that emphasize the functional aspects are process oriented; those that emphasize data base construction are data oriented; those that emphasize temporal aspects are behaviour-oriented methods.
2. Classification is based on the *stages of the development process addressed*. Thus, we have methods for requirements engineering, systems, design, construction design, etc.
3. Classification is done on the extent of use of *mathematical formalisms*. This leads to formal and systemic methods.
4. Classification is based on the *nature of issues handled* in the method. Soft methods like Soft Systems methodology [17] and SISTeM [18] make it possible for different organizational actors to develop a view of the organization and of the information systems needed to support this view. In contrast, hard methods concentrate on the technical aspects of information systems product development and ignore the larger organizational roles played by them.

A number of other classification schemes exist [19–22] and each has its own perspective of classification.

Looking now at meta-models, we find

1. Classification is based on the *treatment given to the product and process aspects* of methods. This leads to product meta-models, process meta-models, and integrated product–process meta-models.
2. Classification is based on the *method feature emphasized*. This yields [13] basic meta-model, guidance meta-model, traceability meta-models, etc.

The foregoing classification schemes aid our understanding of methods and meta-models, respectively. However, we wish to exploit this body of work *to help us in arriving at the set of concepts that could go into defining a generic model*. Meta-model classification (1) tells us that there are two aspects of methods, product and process aspects. Looking at the product aspect, method classification (1) says that products can have different *nature of concepts*. Concepts can be for data, function, event, etc. Classifications (1) and (2) also suggest that method products vary in *complexity*. A final aspect of the product concerns the *nature of the real world phenomena* being captured. Information systems methods are concerned with abstractions of the real world. However, methods in some areas, particularly artificial intelligence, deal with non-abstracted phenomena.

Now consider the process aspect. Method classification (2) suggests that in methods spanning a single stage products are constructed. However, in moving from one stage to the other, a product may have to be transformed to assume another form. Thus, the purpose or *aim* of the method, whether to construct a product or to transform a product, is to be taken into account because it determines the nature of the process steps involved.

The foregoing analysis suggests that four factors of methods are of importance in developing the generic model. These constitute a four-dimensional space in which methods can be placed. We consider these in greater detail below:

- The *complexity* dimension: We postulate that the *complexity of a method* is related to the complexity of the product to be produced. If a product is more complex than another, then the method to produce it must also be more complex. We measure product complexity in terms of product 'facets'. A facet is the set of tightly coupled concepts, whose instantiation results in a product. As the number of facets of a product increases, the complexity of the method increases. The simplest method is the single-facetted one, and we refer to it as atomic. Methods with higher facets are compound. As an example, a method to build an ER product is atomic whereas the Rational method is compound.
- The *aim* dimension: This dimension considers the *purpose behind* the method, what the method tries to achieve. We express method aim in terms of the nature of the product that is to be produced. A product can be a new one, *constructed* from scratch, by reuse, etc. or it can be a *transformation* of another product. Thus, along the *aim* dimension, there can be constructional methods and transformational methods having the purposes to construct a product and transform a product, respectively.
- The *domain* dimension: Here, we consider the *nature of the phenomena* dealt with in the method. We believe that there are two kinds of phenomena of interest, real and abstract. For example, the ER model provides abstraction-based concepts. Thus, methods to build ER products lie in the abstract domain. In contrast real domains are of particular interest in artificial intelligence. As an example consider playing with toy blocks. The specific blocks are given and the player manipulates these real blocks rather than abstractions of these. A specific method for playing with the given blocks can be devised. If the blocks are changed then a new method has to be built and, as before, this method is also specific to the new game.
  Thus, we say that the *nature* of phenomena may be either real or abstract and we get real domains or abstract domains.
- The *semantic* dimension: This dimension deals with the *nature of concepts* of the method. Along this dimension we consider methods to use active or passive concepts. Passive concepts capture passive, data-oriented aspects of products whereas active concepts capture active, process or dynamic, aspects of products.

3 Properties of the generic model

The foregoing suggests that a generic model should have the following properties:

1. The generic model should be *class independent*. That is, it should be possible to represent in the generic model a method that occupies any position in our four-dimensional space.
2. The generic model should be independent of any meta-model. Essentially meta-model independence calls for independence from product meta-models and process meta-models. Interestingly, meta-model independence facilitates class independence as shown below.

Product meta-model independence makes it possible to represent abstract or real domains and active or passive concepts. Thus, it facilitates independence along the *domain* and *semantic* dimensions. Similarly, it facilitates *coverage* independence because products of any stage of the life cycle can be instantiated. Independence along the *aim* dimension is facilitated because the constructed or transformed products can all be handled with product meta-model independence. Product meta-model independence facilitates the representation of one or more sets of concepts to represent one or multiple facets. Thus, independence along *complexity* is also facilitated.
Process meta-model independence facilitates independence along *aim* and *coverage* dimensions. Along the *aim* dimension, different process steps are followed in building constructed or transformed products, respectively, and process meta-model independence facilitates this. Again, *coverage* independence is achieved because process steps of any stage of the life cycle can be handled. Similarly, it can be argued that *domain* and *semantic* dimensions are also facilitated by process meta-model independence.
The foregoing shows the close linkage between product and process aspects of methods and the need to integrate these two aspects together.

3. The three-layer generic framework suggests that the generic model must provide an *essential view* of a method. This view should abstract out the common properties of meta-models. It should reveal the nature of methods, what they are, what they do, and how this capability is achieved.
4. Features of methods that are essential but treated as 'add-ons' today should be *integrated* in the method and represented at a suitably high level of abstraction. Thus, it should be possible to express method constraints, heuristics, guidelines, etc. in the generic model.

4 Towards the generic model

In this section we develop our approach to building the generic model. We shall consider the implications of each property identified in the previous section and identify the basic concepts of the model and their inter-relationships. These shall be put together in the next section to form the generic model itself.

4.1 Class independence

Since class independence calls for meta-model independence, the holistic structure of the artefact, its component parts, and inter-relationships between these components must be expressed in abstractions high enough to instantiate a meta-model. These concepts of the generic model, the generic concepts, must *not* be technical artefacts, graphs, relationships, objects, fragments, chunks and the like. Instead, generic concepts should directly capture the notions of product and process. We propose to do this through the notion of product and process primitives.

A *product primitive* is an atomic product unit. It is the smallest unit that can be operated upon. After a product primitive has been manipulated, it may be mandatory or optional to manipulate another related product primitive. Correspondingly, a *process primitive* is the atomic process step that can be enacted. After a process primitive has been enacted, the enactment of some other process primitive may become mandatory or optional. The generic model should articulate all mandatory and optional consequences of primitive enactment.

The inseparability of the product and process aspects has been mentioned above. To handle this we integrate *process primitives* and *product primitives* together in the notion of *method primitives*. A *method primitive* is the smallest meaningful process unit that can be enacted. It specifies which *process primitive* manipulates which *product primitive* to produce what result.

The application engineer uses the set of *method primitives* to construct the development process. This can be done in two ways, by

- Following a process model or
- Some ad hoc means

The former is a statement of ‘what is to be done when’. Thus, the sequence of method primitives to be enacted is pre-decided and any decision making required to determine what is to be done next is part of the process model. In the ad hoc approach, the decision as to what to do next is taken by the application engineer explicitly, as development proceeds. Thus, the issues around process modelling are external to this approach.

We believe that the ad hoc approach is suited to the development of our generic model. First, it makes the method independent of any process model. As discussed earlier, this facilitates genericity. Second, the enactment mechanism becomes simpler: since no decision making is to be incorporated in it, only a mechanism *for sequential enactment of selected method primitives* is required.

4.2 An essential view

We [12] view a method as an artefact that provides the capability to construct a product. A method has two aspects to it, its static and dynamic properties. Static properties of interest, from a modelling point of view, are those that describe the holistic structure of the artefact, identify the component parts of the artefact, and bring out the inter-relationships between these components. The dynamic properties of artefacts are those that describe the interaction of the artefact with its environment. We believe that there are two kinds of environments relevant to a method (see Fig. 1). The *technical* environment highlights the technical behaviour of the method: the technical input, technical response, technical features to be supported, etc. This environment consists of the product to be developed and the process used to develop it. The *usage* environment looks to the manner in which the method will be used. It deals with its functional aspects: who uses the method, how is it used, what functionality is needed, etc. The *usage* environment is principally determined by two actors, the application engineer and the method engineer.

The technical environment suggests that things that flow across the artefact–environment boundary are

1. Process stimulus
2. The product under development
3. Product information, for example, product quality metric values
4. Guidance information, what can or must be done next
5. Trace information

It follows that the static aspect of methods must contain generic concepts for these. Further, the dynamic aspect is a mechanism that processes the process stimulus and generates (2)–(5) above.

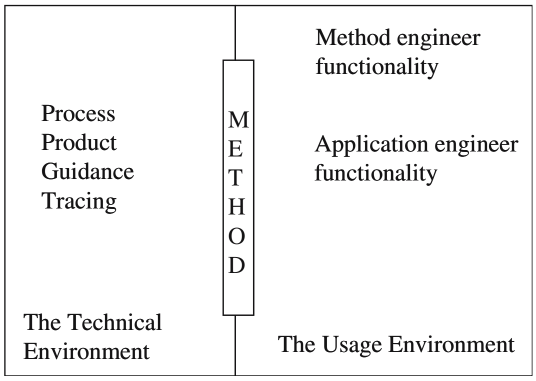

Process
Product
Guidance
Tracing
METHOD
Method engineer functionality
Application engineer functionality
The Technical Environment
The Usage Environment

Fig. 1 The two environments

It can be seen that the technical environment is useful in developing the generic model. It provides knowledge for defining the static and dynamic aspects of the generic model.

The *usage* environment reveals the functional capability that should be built around a method. This capability should include

1. For the method engineer: method instantiation. The three-layer framework suggests that generic model instantiation yields the meta-model whose instantiation, in turn, yields the method. This is the traditional approach to meta-modelling. However, given the generic model, the existence of the meta-model makes sense provided a number of methods can be instantiated from it. When only one method is required, then it would be best if the generic method could be instantiated to directly yield the method. This latter possibility arises, as mentioned above, in the domain of real methods.
   In other words, there should be features to instantiate the method or the meta-model from the generic model. It follows that we need to extend CAME technology to include the former.
2. For the application engineer: capability should be provided to decide the process stimulus, view the product, and obtain all product information. Additionally, support for the development process is needed: features for guidance in deciding what to do next, tracing, backtracking, etc. Reuse of previous experience must be catered to.

4.3 Integrating essential features

As mentioned earlier, there are some essential features of methods like method constraints, heuristics, guidance, tracing and backtracking, etc. Consider the first two of these. First, we require a representation of the constraint and heuristic to be satisfied and we express these as product primitives. Second, we need to define appropriate process primitives for constraint or heuristic satisfaction. We find two different ways in which these primitives could act:

- Primitives *check* for constraint and heuristic satisfaction.
- Primitives *enforce* constraints and heuristics.

Constraint and heuristic *enforcement* is essentially a process model: the product is examined and appropriate method primitives are enacted to modify it. In accordance with the ad hoc approach adopted by us, we reject this option. Thus, we shall define process primitives that *check* constraint and heuristic satisfaction. Just as for other primitives, the application engineer has to decide when to enact these.

As discussed earlier, the enactment of certain method primitives may imply mandatory or optional checking for constraint and heuristic satisfaction. We need an abstraction that models this successor-predecessor relationship. Similarly, to provide guidance we need this abstraction to tell us which method primitives must or can be enacted after which ones. Tracing and backtracking require that this abstraction be traversed in the reverse direction.

5 The generic model

The generic model consists of two complementary parts, the static and the dynamic parts. The *static part* deals with (a) product, process and method primitives, (b) dependencies between method primitives, (c) the relationship between method primitives and a method, and (d) method typology. The *dynamic* part deals with method primitive enactment and the role of dependencies in process modelling, process guidance, tracing and backtracking.

5.1 Generic method statics

In this section we provide an overview of generic statics. Details can be found in [15]. As shown in Fig. 2, we view a method as a collection of one or more *method blocks*. This is shown by the M:N relationship between a *method* and *method block*: a *method block* may belong to one or more methods and a *method* may have one or more *method blocks* in it. The specialization hierarchy of a method shown in Fig. 2 models the complexity and aim dimensions of our classification. In accordance with the complexity dimension, a method can be *atomic* or *compound*. An *atomic* method deals only with those products that are expressed in exactly one product model whereas a compound method is composed of other simpler methods. It is possible to consider the type of a method as constructional or transformational. This is in accordance with the aim dimension. A constructional method is used whenever a new product, expressed in one or more product models, is to be constructed. In contrast a transformational method is used for transforming a product, expressed in or more product models, into a product of other product model(s).

A *constructional* method that builds products for the ER model is *atomic* since the product is expressed in exactly one model. Similarly, the transformational

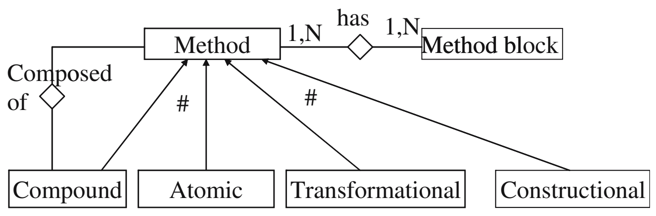


Fig. 2 Generic method statics

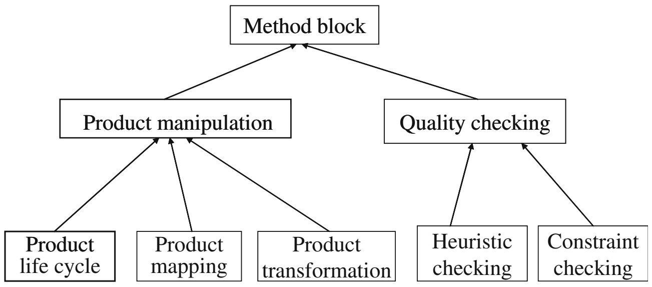


Fig. 3 Types of method blocks

method for converting an ER product into a relational product is atomic since the product to be transformed and the resultant product are both expressed in one product model each. Now consider examples of compound methods. The *constructional atomic* methods to construct products in accordance with the object model, functional model, and dynamic model of OMT, compose the *constructional compound* method of OMT. Similarly, the constructional atomic methods to construct the use case and object schema comprise the rational compound constructional method. Transformational methods may also be compound. For example, a method to convert the rational product to an object-oriented design product is compound: the transformation deals with multiple product models as input even though there is only one product model on its output side.

When expressed in terms of *method blocks*, *compound* methods are composed of *method blocks* that belong to several methods whereas all *method blocks* of an *atomic* method belong to the *atomic* method only.

The different types of method blocks are shown in Fig. 3. Broadly, these are of two kinds, *product manipulation* method blocks do product development and *quality checking* method blocks that check for and provide information about the quality of the product. The expectation is that application engineering is a product development–quality checking–product development cycle.

Figure 3 shows that there are three kinds of product manipulation method blocks. The first, *product life cycle* method blocks pertain to atomic methods. They can (a) create instances of product primitives, (b) relate these instances together, and (c) delete product instances. The second, *product mapping* method blocks pertain to constructional methods and establish a mapping between instances of one component method with those of the other. Thus, for example, an object of an object model can be mapped to a data flow of a functional model. The last, namely, product transformation method blocks pertain to transformational method and cause the transformation of an instance of one model to those of the other. For example, an entity of the ER model could be transformed to a relation of the relational model.

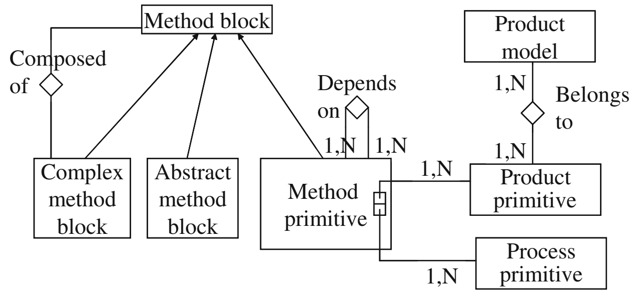


Fig. 4 Structure of a method block

Quality checking method blocks when enacted, supply information about the quality of the product. Heuristic checking blocks tell the application engineer whether a given heuristic has been satisfied or not whereas constraint-checking blocks provide information on the satisfaction of method constraints.

The structure of a method block is shown in Fig. 4. When viewed structurally, it is possible to classify a method block as a *method primitive*, an *abstract method block*, or a *complex method block*. A *method primitive* is a pair, <objective, approach>. The objective of a *method primitive* tells us what it tries to achieve. The approach tells us the technique that can be used to achieve the objective of the block. An objective itself is a pair <product primitive, process primitive>. Product primitives belong to the product model. For each product primitive of the product model, the process primitives that act upon it are associated with it to yield the objectives of the method.

Complex method blocks are composed from simpler method blocks. For example, the complex method block, build aggregate, can be defined to build an aggregate entity from primitive method blocks to create entities and to establish relationships between them. *Abstract method blocks* are built to generalize method blocks based on their common properties. For example, two *complex method blocks*, build aggregate and build ISA hierarchy can be abstracted into a method block, build hierarchies based on the common property that both these construct hierarchies, aggregate and ISA hierarchies, respectively.

Method primitives exhibit dependencies among themselves. This is shown by the *Depends on* relationship in Fig. 4. Dependencies are statements about which method primitives must or can be enacted after a given method primitive is enacted. Clearly, dependencies get carried over to complex and abstract method blocks and therefore we can speak of dependencies between method blocks and not just method primitives.

We will now consider the main concepts of Fig. 4 in detail.

5.1.1 *Product primitives*

Methods have concentrated on producing products that are well defined and well formed. As a consequence, a method focuses on the concepts of the product model, and ensuring that the product satisfies the tenets of the model. From this perspective, one can find product primitives by a detailed examination of the product model.

We classify product primitives into two, product *manipulation primitives* and product *quality primitives*. The former include concepts and their inter-relationships whereas the latter include quality concepts like method constraints and heuristics.

In an atomic method there is only one product model and the foregoing suffices. However, in compound methods, consistency across every pair of product models has, additionally, to be ensured so that the compound product is properly composed of its simpler products. This composition requires product composition primitives and composition quality primitives.

It can be seen that our product primitives are at the required level of abstraction. They are independent of any product model or meta-model. A product primitive may be instantiated to any product model or meta-model concept, an abstract one (graph, an object, an entity) or a real one (a sound clip, a block of a toy). Thus, domain independence is protected. Product primitives make no assumption about the nature of the product that may belong to any stage of the life cycle. Thus, the coverage dimension is promoted. As the complexity of the method rises, the set of product primitives becomes large. While a way of handling this large number has to be found, product primitives do not in any way inhibit the construction of methods of any complexity or of different aims.

5.1.2 *Process and method primitives*

A process primitive is the smallest unit of process. It operates on a product to cause a change in it thereby producing a new product. Since constructional and transformational methods perform different operations on the product, process primitives for these are different from one another. Consider *atomic methods* first. Constructional methods need to construct individual product concepts, inter-relationships between these and provide information about product quality. Thus, we have three types of process primitives as follows:

1. Create or delete instances of product primitives.
2. Establish inter-relationships between instances of product primitives.
3. Display product information on demand: is the product complete? Is it consistent? Does it comply with heuristics?

For a transformational method the process primitives are those that transform from the source to the target product model. Therefore we need essentially one kind of process primitive to transform instances of different kinds

1. An instance of the source product primitive to the target product primitive.
2. An instance of the source inter-relationship to the target one.
3. An instance of the source quality primitive to the target one.

Now consider compound methods. We need process primitives for mapping. This yields the product composition class and the composition quality class of process primitives.

It can be seen that the taxonomy of method blocks shown in Fig. 3 draws heavily from the different kinds of process primitives described above.

5.1.3 *Dependencies*

As shown in Fig. 4, method primitives are dependent upon one another. We postulate that there are different ways in which method primitives are dependent upon one another. This is captured in the notion of a dependency type. For example, in the meta-model of [15] there are four dependency types called requirements, removal, activate and inactivate, respectively.

A dependency shows up in two forms

1. The enactment of a primitive may call for the enactment of another, or
2. Primitive enactment may call for another primitive *not* to be enacted.

We refer to the former as an additive dependency whereas the latter is a subtractive one. More formally, let us be given a set of dependency types, $DT = \{DT_1, DT_2, \ldots, DT_n\}$. For an *additive dependency* we say that a method primitive O_2 is dependent upon a method primitive O_1 under a dependency type DT_i if the enactment of O_1 is a pre-requisite for the enactment of O_2 under that dependency type. As an example, let there be a dependency type, trigger. Then $O_1 \rightarrow O_2$ (O_2 depends on O_1) says that for O_2 to be triggered, the enactment of O_1 is a pre-requisite. In other words, the enactment of O_1 makes O_2 available for enactment. For a *subtractive dependency*, a method primitive O_2 is dependent upon a method primitive O_1 under a dependency type DT_i if the enactment of O_1 results in barring the enactment of O_2 under that dependency type. As an example, let there be a dependency type, withdraw. Then $O_1 \rightarrow O_2$ says that the enactment of O_1 makes O_2 unavailable for enactment.

Two attributes, urgency and necessity, are associated with each dependency type. Urgency refers to the time at which the dependent method primitive, O_2, is to be enacted. If O_2 is to be enacted immediately after O_1 is enacted then this attribute takes on the value *Immediate*. If O_2 can be enacted any time, immediately or at any

Table 1 Nature of dependencies

Type	Urgency	Necessity
1	Immediate	Must
2	Immediate	Can
3	Deferred	Must
4	Deferred	Can

moment, after O_1 has been enacted, then urgency takes on the value *deferred*. Necessity refers to whether or not the dependent method primitive O_2 is necessarily to be enacted after O_1 has been enacted. If it is necessary to enact O_2, then this attribute takes the value *must* otherwise it has the value *can*.

These attributes provide the basic properties to be satisfied by method primitives related to each other under a dependency type. Four cases arise as shown in Table 1.

Dependency type 1 of the table says that the dependent method primitive O_2 must be enacted immediately after O_1 has completed. Dependency type 2 says that O_2 can be optionally enacted; and if it is in fact enacted, then this must be done immediately upon the completion of O_1. A delayed enactment of O_2 is disallowed. Dependency type 3 says that O_2 must be enacted after O_1 has been enacted but this may be done at any moment after the completion of O_1. Finally, dependency type 4 says that O_2 may be optionally enacted any time after the completion of O_1.

Consider the meta-model of [15]. In this meta-model four dependency types have been defined, requirements dependency, its inverse called removal, activate dependency, and its inverse called inactivate. Requirements dependency says that a dependent method primitive O_2 must be enacted after O_1 but it may be enacted any time after O_1 is completed. This corresponds to dependency type 3. Its inverse dependency, removal is of type 1. It says that O_2 must be immediately removed from the set of method primitives to be enacted after O_1 is complete. The activate dependency is of type 4. It says that O_2 may be optionally enacted at any later moment after the enactment of O_1 is over. Its inverse, inactivate dependency is of type 1. The method primitive O_2 must be immediately inactivated if it was activated by an earlier dependency. It can be seen that in the example meta-model, there is no type 2 dependency.

It can be seen that requirements and activate are additive dependency types whereas removal and inactivate are subtractive ones.

5.1.3.1 The dependency graph It is possible to organize method primitives in a directed dependency graph for each dependency type. Thus, there are as many graphs as the number of dependency types in a method. The method primitives in a dependency graph that have no edges entering them are particularly interesting. We refer to these as ‘start’ method primitives. Let the set of start method primitives under a dependency type DT_i be denoted by S_i. We can compute the set of start method primitives START, for the entire method by determining the common start method primitives across all dependency type graphs:

$$\text{START} = \cap S_i \text{ for all } i.$$

Similarly the dependency graph has nodes that have no outgoing edges. Just as the former are start method primitives, the latter are stop method primitives. After their enactment, the development can halt. Just as for start method primitives, the set of stop method primitives is obtained by taking the intersection of the stop method primitives of each of the dependency graphs. Let this set be STOP.

START is interesting because it identifies those method primitives that can be used to start off application development. Thereafter, the dependency graphs can be followed to select the method primitives to be enacted. The development process can be halted when a method primitive belonging to STOP has been enacted.

Consider a method with the set of method primitives $O = \{O_1, O_2,...,O_{14}\}$. Let there be two dependency types DT_1 of type 1 and DT_2 of type 2. Let the following dependencies be defined:

DT_1 dependencies

$$\begin{array}{llll} O_1 \to O_2 & O_1 \to O_3 & O_1 \to O_4 & O_1 \to O_5 \\ O_6 \to O_7 & O_6 \to O_8 & & \\ O_9 \to O_{10} & O_9 \to O_{11} & O_9 \to O_{11} & \end{array}.$$

It can immediately be seen that method primitives with no edge incident on them are O_1, O_6, O_9. Therefore, $S_1 = \{O_1, O_6, O_9\}$. The dependency graph is shown in Fig. 5.

DT_2 dependencies

$$\begin{array}{ll} O_1 \to O_6 & O_1 \to O_9 \\ O_6 \to O_{13} & O_6 \to O_{14} \end{array}.$$

Again, it can be seen that O_1 is the only method primitive with no edge incident on it. Therefore, $S_2 = \{O_1\}$. The dependency graph is shown in Fig. 6 with this kind of dependency being shown by a dotted arrow.

We can put the two graphs of Figs. 5 and 6 together to yield the total dependency graph for our method as shown in Fig. 7. The set of start method primitives, START = $S_1 \cap S_2$ = $\{O_1\}$. It can also be seen that STOP = $\{O_2, O_3, O_4, O_5, O_7, O_8, O_{10}, O_{11}, O_{12}, O_{13}, O_{14}\}$.

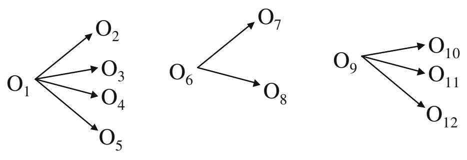


Fig. 5 The dependency graph under DT_1

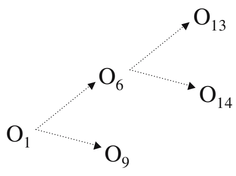


Fig. 6 The dependency graph under DT_2

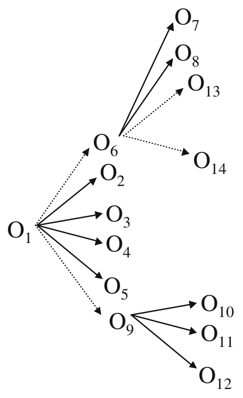


Fig. 7 The dependency graph for the method

It is to be noted that whereas a dependency graph highlights the dependency structure of method primitives, it does not make explicit the additive–subtractive or the necessity–urgency properties of a dependency type. These are to be determined from the nature of the dependency type itself.

In Sect. 5.2 we will see that dependency graphs can be used to provide support for guidance, tracing, and backtracking.

5.2 The generic model: the dynamic part

Whereas static aspects of a method deal with the structure of a method, method dynamics deals with the behaviour of methods. It determines the response of a method to stimuli provided by application engineers. In this stimulus–response view, the development process is a sequence of stimuli selected by the application engineer for enactment. The algorithmic process, the enactment algorithm, that responds to this sequence constitutes method dynamics.

The stimuli of interest are

1. Requests for enactment of product manipulation method blocks
2. Requests for enactment of product quality checking method blocks
3. Requests for guidance [23] in method block selection
4. Requests for trace information
5. Backtracking requests

We summarize the stimulus and the corresponding response in Table 2.

A stimulus has two components (a) a product part and (b) a method block. It says that the product part must be manipulated by the method block. The product part is found in the product under development and is that piece of the product that requires application programmer attention. The product part must have the same number and type of arguments as the method block. It is only when this match occurs that the method block can operate upon the product part.

Table 2 Method dynamics

Stimulus	Response
Product manipulation	Product after enactment
Quality checking	Quality metric value
Guidance seeking	Menu of method blocks that can be enacted next
Tracing up to N previous enacted method blocks	Up to Nth previous enacted method blocks in reverse order of their enactment
Backtracking to Mth previously enacted method block	Product at backtrack point Enacted method block at backtrack point

5.2.1 *Product part*

We define a product part as that part of the product, which is the focus of attention for further exploration. It is axiomatically defined as follows

- Every *instance of a product primitive* is a product part. For example, in the ER model, an attribute, an entity and a relationship type are product primitives. Instances of these like date_of_birth, employee, and employs, respectively, are product parts.
- Every set of instances of product primitives participating in a *method primitive* is a product part. For example, the set of entity type instances that are arguments to the method primitive to construct a relationship is a product part.
- Every set of instances of product primitives participating in a *complex method block* is a product part. For example, the set of entity type instances that are arguments to the complex method block, *build aggregate* of Sect. 5.1, is a product part.
- Every set of instances of product primitives participating in an *abstract method block* is a product part. Again, the set of instances of entity types participating in *build hierarchies* of Sect. 5.1 constitute a product part.
- There is a *special product part* called ‘don’t care’. Every product has it. It says that irrespective of what the product is (‘don’t care’ what the product is) some process primitives can be used. For example, the primitive *create* can be applied to ‘don’t care’ to create new product parts.

5.2.2 *The development process*

The development process is a sequence of development process constructs (DPC). A DPC represents the stimulus and consists of a product part and a method block. We define it as an objectified relationship between product part and method block

$$\mathrm{DPC} = ([\mathrm{pp}], \mathrm{mc}),$$

where pp is the product part, mc belongs to the set of method blocks, and mc can manipulate pp. The following are examples of DPCs

([don't care], < entity, create >)
([date_of_birth, employee], < attribute, entity, couple >).

The first says that the method block to create an entity can be used on the product part *don't care* to create a new, named entity instance whereas the second says that the method block to couple an attribute and entity can be used to couple date_of_birth and employee to each other. At the end of enactment of this DPC, employee has an attribute, date_of_birth.

Our development process deals with instances of method blocks. These must be in accordance with the dependency graphs of a method. Initially, the application engineer selects an instance of a method block from START (see Sect. 5.1.3.1). The enactment algorithm enacts it and the product is accordingly modified. The next method block instance that can or must be enacted is obtained by following the dependency links in the various graphs. Since the type of the enacted method block is known, the dependency graphs yield the set of method block types whose instances can now be enacted. In this way, it can be seen that any development process is a *route* among the dependency graphs and that the only processes that can be built are the ones defined under the dependency graphs of the method.

Let us denote the *j*th instance of method block O_i by $o_{i,j}$ and show the progress of a typical development process. Let the development process start by enactment of $o_{1,1}$. Now, the dependency graph of Fig. 5 suggests that instances of O_2–O_5 could be enacted under DT_1 whereas, by Fig. 6, O_6 and O_9 could be enacted under DT_2. Let the application engineer enact $o_{6,1}$. Following Fig. 5, it is now possible to enact instances of O_7 and O_8 under DT_1 as well as O_{13} and O_{14} under DT_2. Let the application engineer select $o_{7,1}$ for enactment. The dependency graph now suggests nothing. The application engineer may choose to terminate the process at this point or may start with another instance of O_1.

Assuming that the process is terminated, we get the following process

$o_{1,1}$
$o_{6,1}$.
$o_{7,1}$

Equivalently, the process (see Fig. 8) is the route taken in the dependency graph of Fig. 7.

Supporting the development process To support the development process outlined above, we develop an enactment algorithm. This algorithm must provide capability to handle

- Product manipulation primitives
- Quality checking primitives
- Guidance in selecting the next manipulation or quality checking primitive to be enacted
- Tracing and backtracking

We first lay out our policy for guidance, backtracking, traceability, and quality checking and then present the full enactment algorithm.

5.2.3 *Providing guidance*

As is well known, there are two approaches to guidance, the passive and the active. We adopt the former approach here. This approach says that (a) upon enactment of a method block the set of method blocks that can be enacted next is calculated, (b) the application engineer is provided this set to select the next method block to be enacted.

We develop three kinds of guidance here

- Basic guidance
- Operational guidance
- Semantic guidance

The set of dependency graphs of a method defines the entire *basic* guidance capability available in the generic model. Since progression from one method block instance to the next must follow the directed edges of these graphs, the enactment of a method block should initiate the determination of its successor nodes. Thus, the method blocks that can be enacted next can be determined and presented to the application engineer. *Operational* guidance is built on top of basic guidance (see Fig. 9) and considers operational behaviour patterns of application engineers. These patterns form the basis of partitioning the set of method blocks determined by *basic* guidance into sets specific to a behaviour pattern. The partitions can be presented to application engineers.

Semantic guidance is also built on top of basic guidance (Fig. 9). However, the basis of classification is now different from that of operational guidance. We are

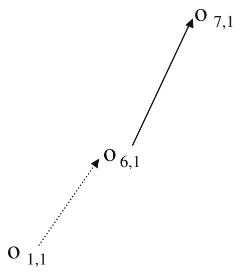


Fig. 8 A dependency graph route

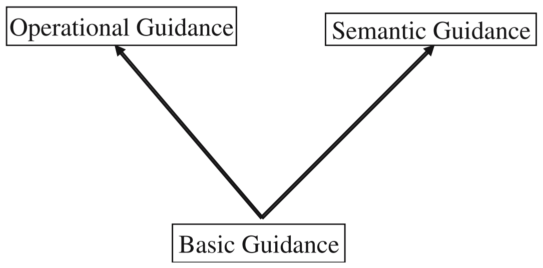


Fig. 9 Guidance levels

now *not looking for operational behaviour* patterns but for the *intention* that the application engineer has in mind. The set of method blocks obtained through basic guidance are partitioned according to the *intention* that they help in fulfilling. Application engineers are then presented with only those intentional sets that are relevant to the task at hand.

We now consider each kind of guidance in detail.

5.2.3.1 Basic guidance level Consider that the development process is just starting out. The application engineer may want to know the objectives that can be enacted to start off. The guidance mechanism must display the set START to enable selection of an objective to be enacted. Once an objective has been selected, guidance is provided by following the dependency graphs. Given a node that has been enacted, the graphs are traversed along the direction of the edges. The successor nodes (those that are one edge removed in the direction of traversal) of the given node for each graph are determined, the additive–subtractive nature of dependency types is taken into account and the set of objectives available for enactment is calculated. This is the menu from which the next objective to be enacted can be selected.

Notice that we do not take into account the urgency and necessity properties of dependencies. This is because our intention is to present a menu that identifies what can be done next rather than when it must be done (urgency) or whether it must or can be done (necessity).

We formalize the foregoing now. Let there be a set G_i, called the choice set, which contains the set of method blocks that can be enacted next after method block instance d_i has been enacted. Under the passive guidance approach, G_i is the set of method blocks that is to be provided to the application engineer by the guidance mechanism. Initially G_0 = START. When d_i is enacted then the guidance mechanism computes G_i as follows:

1. For each graph instance in which $\mathrm{d_i}$ occurs, find the corresponding dependency graph type
2. For each such graph type

For each directed edge going out from the method block type corresponding to d_i

Find the successor node, N.
If dependency is additive then $G_i = G_i \cup N$ but if it is subtractive then

$G_i = G_i - N$.

It can be seen that as each method block instance is enacted, G_i is modified to reflect the new method blocks that can possibly be selected.

Consider providing guidance for the process of Fig. 8. At the start, the guidance mechanism presents O_1 as the only objective from which development process is to begin. After enactment of $o_{1,1}$ the guidance mechanism determines that its type is O_1. The graphs DT_1 and DT_2 are traversed from O_1. The former yields the set $\{O_{2,} O_3\, O_{4,} O_5\}$ and the latter yields $\{O_{6,} O_9\}$. Thus, the guidance mechanism suggests the set $\{O_{2,} O_3\, O_{4,} O_5, O_{6,} O_9\}$. The application engineer now selects the objective O_6 and proceeds to enact its instance $o_{6,1.}$ The guidance mechanism determines the sets $\{O_{7,} O_8\}$ and $\{O_{13,} O_{14}\}$. Additionally, taking into account the self-loop on O_1, the set presented is $\{O_{2,} O_3\, O_{4,} O_5, O_6, O_{7,} O_{8,} O_{9,} O_{13,} O_{14}\}$. The developer selects an instance $o_{7,1}$ of O_7 for enactment. This enactment leads to no further addition in the choice set and O_7 is the terminating node in the graph. Application development may now stop, if the product meets application needs.

5.2.3.2 Operational guidance level The choice set G contains all possible method blocks that can be enacted after the enactment of a method block instance. From the previous section it can be surmised that that the choice set G_i grows very rapidly and can become extremely large. It has been shown in [12] that this large size makes it very difficult for the application engineer to select a meaningful method block.

To obviate this difficulty, operational guidance exploits typical patterns of an application engineer's operational behaviour. In computer science such operational behaviour patterns abound. For example, in operating systems, page replacement algorithms use LRU LFU patterns to determine the page to be discarded. Similarly, in programming, it is assumed that flow of control is sequential. Computers are organized to increment the program counter by unity under the assumption that the next sequential instruction shall be executed. In a similar way, it is possible to postulate the *localization* pattern: the application engineer may like to concentrate on the *most recently manipulated product part*. Localization is so named because attention is on the development of a small, localized part of the product.

Let the most recently enacted DPC be d. Two cases arise as follows:

1. d is unary. Its enactment may result in either a product part 'p' or a 'don't care' being produced. Localization says that the next DPC may be either another unary DPC operating on p or 'don't care', or an n-ary method block having p as one of its arguments.
2. If d is n-ary then the next DPC may have any one or more arguments of d as its arguments, in addition to 'don't care'.

Case (1) continues to maintain focus on the products of d whereas case (2) provides the possibility of a gradual movement away from earlier products.

Consider operational guidance in a small part of a development process as follows. An entity, employee, is created by a unary DPC, ([don't care], <entity, create>). The operational guidance mechanism computes the choice sets for, among other product parts, 'don't care' and 'employee'. The application engineer chooses to focus on employee, looks at the employee specific choice set and decides to define an attribute for employee. A binary DPC is used to define 'name' as an attribute of 'employee', ([name, employee], $<$entity, attribute, couple$>$). The choice set is computed for all part parts including for name and employee. By (2) above, it is possible to choose either employee or name as the next product part to be developed. Focusing on name causes a shift away from employee. Notice how attentions shift from 'don't care' initially, to 'employee and then to 'name'.

Of course, it is possible that an altogether different product part becomes interesting suddenly, calling for a clean break from the localization pattern. For example, after making 'name' to be an attribute of 'employee', the application engineer may want to create a new entity type 'department'. This is supported by the operational guidance mechanism by the computation of the choice set specific to 'don't care'.

Let us now consider the guidance mechanism to support operational guidance. Basically, the mechanism needs to determine the method blocks that are applicable to each product part comprising the product. Recall that 'don't care' is a product part of every product. The localization pattern implies that the guidance mechanism must present only those method blocks that are relevant to the localized task at hand. Thus, the choice set G_i is to be partitioned product part wise. Given product parts P_1, P_2,...,P_n, to obtain their localized choice sets G_{i1}, G_{i2},...,G_{in}, respectively, the following action is taken

1. For unary d_i: Determine the product part produced by d_i. This can be either 'don't care' or an instance of a product type, P_j. The application engineer chooses the product part to be developed further. The choice set corresponding to this is presented.
2. For n-ary d_i: Determine the product parts operated upon, say P_1, P_2,...,P_n. Then any of G_{i1}, G_{i2},...,G_{in} may need to be presented.

Once the partitioned choice sets are available, it is possible for the application engineer to select the product part of interest through any operational pattern (most recently manipulated, most frequently manipulated) and ask for the appropriate partition.

Method dynamics can now be visualized as a cycle shown in Fig. 10. At each turn of this cycle, the enactment mechanism creates localized choice sets for use by the operational guidance mechanism.

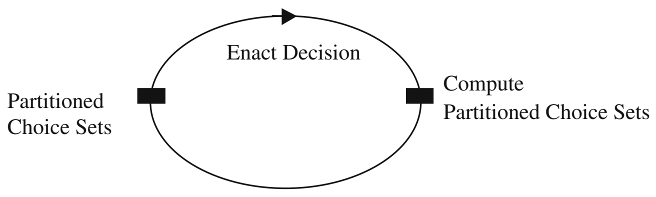


Fig. 10 The enactment mechanism

5.2.3.3 Semantic guidance Whereas operational guidance makes assumptions about the operational behaviour of application engineers, semantic guidance takes into account the developmental aim that the application engineer wants to satisfy. Only that set of method blocks which helps in achieving this aim is presented to the application engineer. Thus, if the aim of the application engineer is to assess the quality of the product, then only the method blocks that help in quality assessment are presented. On the other hand, if the aim is to develop the product further then method blocks for product development are presented.

Semantic guidance requires a classification of the set of method block types according to the aim that each class fulfils. Let us be given developmental aims A_1, A_2,...,A_n. For each A_i we create a class of method blocks such that each method block fulfils the aim A_i:

$$A_i = \{D_j | D_j \text{ fulfils } A_i\}.$$

For example, a method may have the two basic aims (among others), enforce method constraints and check heuristics. Then, two classes of method block types are produced, the former containing method block types that enforce method constraints and the latter containing method block types that check for heuristic satisfaction.

Developmental aims may be aggregates of simpler developmental aims. As before, there is a class of method block types for the aggregate aim and it consists of all those method block types that fulfil the aggregate aim. The aggregate class is the union of the classes of the simpler developmental aims. Formally, if AA is an aggregate of simpler aims A_1, A_2,...,A_n then:

$$AA = A_1 \cup A_2 \cup \cdots \cup A_n.$$

In case an aim is an aggregate of others then the cycle corresponding to the aggregate aim can be decomposed into a bundle of cycles of lower aims. This allows the guidance mechanism to operate in a hierarchical manner: the developmental aims of the application engineer can be ascertained starting from broad, fuzzy, aggregate aims and going down to increasingly precise aims. Once the most precise aim has been obtained, the guidance mechanism presents the choice set corresponding to this aim. Let there be an aim, enforce product quality that is an aggregate of enforce method constraints and check heuristics. Then the method block types of both these have the aggregate aim of enforcing product quality. The set of method blocks for enforce product quality is the union of the sets of these two.

Method dynamics can now be visualized as multiple alternative cycles as shown in Fig. 11. Each cycle corresponds to a developmental aim of the method. After the enactment of a method block the application engineer has the choice of following one of several developmental aims. The various choice sets are computed upon method block instance enactment. Thereafter,

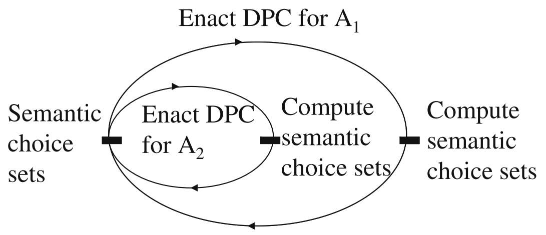


Fig 11 The dependency cycle

depending upon the developmental aim, the appropriate choice set is presented to the application engineer.

5.2.4 *Backtracking*

Backtracking is the ability to go back to an earlier process state. This can happen for two reasons

1. There is dissatisfaction with the product. However, the product was good up to a DPC enacted earlier. Therefore, the product and process has to be restored to this state and a new development route is to be followed. We refer to this as backtracking in the *jettison* mode.
2. There is no real dissatisfaction with the product. However, the application engineer wants to explore another development option starting from some earlier state. We refer to this as *exploratory* mode backtracking.

In terms of dependency graphs, backtracking can be understood to be the inverse of guidance: whereas guidance deals with 'next nodes', backtracking is concerned with 'previous nodes'.

Let the most recently enacted DPC be DPC_n = ([pp_n], mb_n). Let it be desired to backtrack to DPC_k = ([pp_k], mb_k). Then, the enactment mechanism must undoDPC starting from DPC_n till DPC_k is reached. The mechanism to handle this is as follows:

Step 1: If backtracking is in the *exploratory* mode then save the current product, process, and all dependency graph data
Step 2: Repeat till DPC_k is reached

Step 2(a): Restore to [pp_{n-1}] by enacting the inverse of mb_n.
Step 2(b): Move back in dependency graphs to the predecessor node mb_{n-1} of mb_n.
Step 2(c): Discard the node DPC_n from the process and establish DPC_{n-1} as the last process node enacted

As an example of backtracking consider the development process of Fig. 8. Let backtracking occur in exploratory mode after $o_{7,1}$ has been enacted. First the current product, dependency graph data, and the process is saved. Now, let I(mbn) refer to the inverse of method block mbn. Then, as per step 2(a) above, the DPC ([ppn], I($o_{7,1}$)) is enacted. This restores the product to [pp_{n-1}].

5.2.5 *Providing traceability*

Traceability uncovers the process history that led to the enactment of a DPC. Equivalently, it tells the application engineer the evolution history of the product, that is, how did the product evolve to reach its current form. Traceability is similar to backtracking in that interest is in previous nodes of dependency graphs. However, traceability is different from backtracking in its purpose. When backtracking, the aim is to restore development status to a previous product and process state and to proceed with the development process from this restored point. On the other hand, when tracing, interest is in knowing previous development steps. Thus, traceability provides a re-run of the DPCs that were enacted in the past. No restoring of the product or process to an earlier development status is required.

The generic model provides traceability in two modes, burst or decrement mode. *Decrement* mode traceability assumes that the application engineer wants to examine evolution step by step as it occurred. In *burst* mode traceability, the application engineer wants to examine evolution from some *p*th earlier DPC.

To provide traceability, the complete process is saved as for backtracking. However, it is not required to save dependency graph instances. Let the most recently enacted DPC be DPC_n. In *decrement* mode, DPC_{n-1} is displayed and position established at it for supporting further traceability. Further invocation of *decrement* mode traceability causes DPC_{n-2} to be displayed. In *burst* mode, DPC_{n-1}, DPC_{n-2},..., etc. are displayed until the *p*th earlier DPC is reached.

In our example process of Fig. 8, decrement mode traceability will cause $o_{6,1}$ to be displayed. If asked for again then $o_{1,1}$ shall be displayed. If instead of decrement mode traceability is used to trace back two steps, then first $o_{6,1}$ and then $o_{1,1}$ shall be displayed.

5.2.6 *Quality checking*

In Sect. 5.2 we had postulated quality checking method blocks. When these are enacted the application engineer is provided information about product quality. The basics of our quality checking approach are found in the representation theory of measurement of software engineering. According to this theory, entities have quality attributes that are measured through metrics. For example, a software product is an entity. It has attributes size and complexity. The former is measured using the lines of code metric. Similarly, the latter is measured using the order *O* metric.

We extend this to include IS products and their quality attributes. Consider for example the generalization–specialization hierarchy. It has an attribute called depth that can have an associated metric with it. The depth of the hierarchy is controlled though a depth heuristic that, in the case of OMT, suggests that it be restricted to about five levels deep. A number of method constraints are specified in methods. There are four [6] such: completeness, conformity, fidelity and consistency. These are imposed on IS products and can be treated as attributes of these products. Again, we associate metrics with this.

The association of metrics with IS products makes it possible for us [24] to compute the metric value and return to the application an indication of whether the product parts of interest *satisfy* or *do not satisfy* the heuristic or constraint attributes. For this purpose we define quality method primitives as indicated earlier. As examples of these consider

$$<\text{entity, check completeness}>$$
$$<\text{ISA hierarchy, check depth heuristic}> .$$

These primitives participate in a dependency structure with others. For example, since the creation of an entity, employee, calls for its completeness check, we have

$$<\text{entity, create}> \rightarrow <\text{entity, check completeness}> .$$

This is a type 3 dependency in terms of Table 1: it *must* be checked that a created entity is complete but this check can be *deferred* till an appropriate moment.

Since the enactment of a quality method primitive gives information about whether the quality of the product part is satisfactory or not, it is possible for the application engineer to decide which method primitive to enact next.

5.2.7 The enactment algorithm

At the heart of method dynamics is the enactment algorithm that embodies in it support for process enactment, guidance, backtracking and traceability. Thus, it provides all the features that we have discussed so far in this section.

The algorithm assumes that application engineers have a number of different courses of action available and select one of these. These courses of action are represented in the algorithm as the set, possibility. Possibility contains in it the set of DPCs that are available for enactment. Under the passive guidance scheme outlined earlier, this set may be displayed depending upon the guidance mode: basic, operational, semantic or hybrid. The application engineer makes a selection from Possibility and the selected DPC is enacted. This leads to a modification of Possibility. In other words, given possibility$_{i-1}$, a DPC is enacted and this results in possibility$_i$.

With each enacted DPC, three cases arise. The application engineer

1. Makes a fresh selection from possibility$_i$
2. Desires to backtrack in either exploratory or jettison mode, or
3. Wants to trace the development activity so far

In order to handle (2) and (3) above, we introduce special method blocks as follows:

1. < 'don't care', jback > for backtracking in the jettison mode
2. < 'don't care', eback > for backtracking in the exploratory mode
3. < 'don't care', trace(P) > for tracing. P specifies the size of the burst mode trace. By default, trace is decrement mode only

Define possibility, P_i to be the set of all DPCs that are available for selection at a given moment and S_i is the selected DPC from among these for enactment. P_{i+1} is computed by considering the instances of the dependency graphs and by determining the contributions made by the dependencies:

1. The additive–subtractive nature of dependencies tells us the DPCs that become eligible or ineligible for enactment. This suggests two sets ADD(S_i) and SUB(S_i). The former identifies all the dependent DPCs that become *available* on account of additive dependencies, due to enactment S_i. The latter identifies the dependent DPCs that must be made *unavailable* on account of subtractive dependencies due to S_i.
2. The necessity property of dependencies tells us whether additive DPCs are to be mandatory or optionally enacted. This suggests that we partition ADD(S_i) into two sets MAND(S_i) and OPT(S_i) such that

   $$\text{ADD}(S_i) = \text{MAND}(S_i) \cup \text{OPT}(S_i).$$

3. The urgency property tells us the time of enactment of a dependent DPC. Thus, it does not make a contribution to P_i.

Let there be two sets, READY(S_i) and PENDING(S_i). The former computes the set of optional DPCs after the enactment of S_i and the latter, the set of mandatory ones. Then

$$\text{READY}(S_i) = \text{READY}(S_{i-1}) \cup \text{OPT}(S_i) - \text{SUB}(S_i)$$

$$\text{PENDING}(S_i) = \text{PENDING}(S_{i-1}) \cup \text{MAND}(S_i) - \text{SUB}(S_i).$$

Putting together the various components, possibility, P_{i+1} is defined as

$$P_{i+1} = \text{READY}(S_i) \cup \text{PENDING}(S_i).$$

Note that when the development process starts out then the initial $P_i = P_0 =$ START. We start with a null product. Consequently, the only product part available is 'don't care'.

It can be seen that the enactment algorithm supports the dynamic properties of our generic model. All three kinds of guidance, namely, basic, operational, semantic, and hybrid are supported by the foregoing algorithm as follows:

- The set P_i contains in it the set of all DPCs that can be enacted after S_i. This corresponds to the basic guidance level introduced earlier.
- By partitioning P_i part-wise into APSETs we are able to provide support for operational guidance. It the application engineer determines the product part of interest then it is possible to display all applicable DPCs.
- The partitioning of P_i aim-wise makes available the set of DPCs that meet a given aim. The partitioning of each such set into AIMAPSETs provides support for hybrid guidance.

Backtracking is supported naturally due to specific method blocks for its different forms. Since these operate on the product part 'don't care', backtracking can be invoked anytime in the development process. Support for backtracking is described in Sect. 5.2.5.

Traceability is supported through the introduction of specific method blocks that operate on 'don't care'. Traceability can be invoked any time in the development process and the support needed for this is described in Sect. 5.2.6.

6 Conclusion

We have shown in [16] that it is possible to use the generic model to engineer meta-models. Additionally we have shown in [25] that it can be used for engineering 'application-centric' methods as well. For reasons of space, we shall deal with the meta-model and method engineering mechanisms associated with the generic model in a separate paper.

A prototype generic-CAME tool for the generic model has been implemented by a group of students. A more robust implementation is currently ongoing. This generic-CAME tool shall provide an integrated environment for meta-model and method engineering. We intend to use it to do engineering of (a) meta-models like GOPRR (b) methods for real businesses as well as (c) applications in artificial intelligence like playing games with blocks.

Our work with the generic model has so far been centred around 'hard' methods. We have been concerned with the technical issue of building products and have not considered 'soft' methods.

To our knowledge, the only other work that uses the term 'generic model' in the context of methods is [26]. It develops an assembly of ways of doing situational method engineering and a selection from this assembly is to be made. Genericity exists in the sense that the collection of methods forming the assembly can all be used to carry out the same common intention, that of doing situational method engineering. Thus, the intention of the assembly is shared by or is typical of a whole group of methods. This meets the criterion for it to be generic.

It is to be seen that our generic model is not an assembly of methods. It is a system of concepts and inter-relationships, it is a model. It is not possible to select a method in our case. Rather meta-models and methods can be engineered from it. In the assembly approach, the method has already been engineered and is available, as part of the assembly, for selection.

Appendix 1

We show here an example of the use of the generic model by instantiating it with the meta-model of [12]. For reasons of space, we assume knowledge of this meta-model and shall not explain its concepts. The focus shall be on giving a flavour of the way the generic model can be instantiated to yield the meta-model. Thereafter the CAME tool, MERU, reported in [25] can be used to engineer the method itself.

The static part of the generic model is instantiated to yield meta-model statics, instances of method blocks and the set of dependencies between these. In our meta-model, method blocks are instantiated to purposes, product primitives to conceptual structures and fixed structures, and process primitives to operations which can be of two types, product manipulation and quality checking. There are four dependency types, requirements, removal, activate and inactivate. The set of dependencies is instantiated with dependencies between purposes of different types.

Through the click and drag features of the generic-CAME tool, the instantiations of product primitives with conceptual and fixed structures shown in Figs. 12 and 13 are arrived at. As shown in Fig. 12, conceptual structures can be partitioned into two clusters. The first cluster

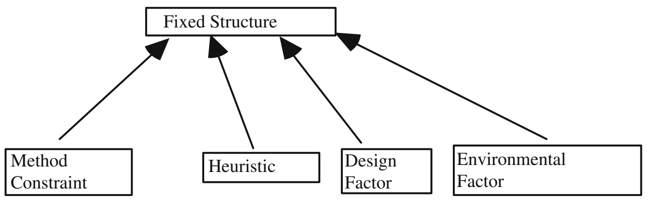


Fig. 13 The fixed structure

Fig. 12 The conceptual structure

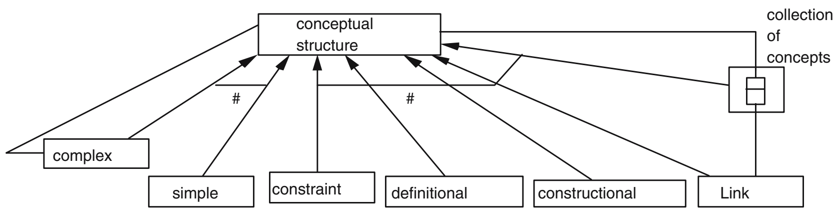
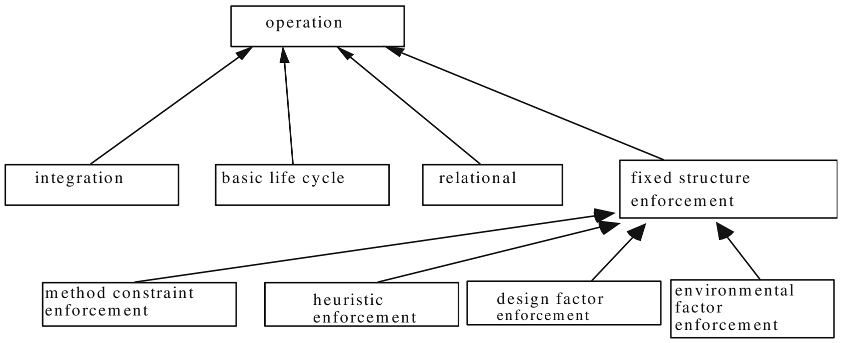


Fig. 14 The operation


operation
integration
basic life cycle
relational
fixed structure enforcement
method constraint enforcement
heuristic enforcement
design factor enforcement
environmental factor enforcement

classifies them as either simple or complex. The second cluster partitions conceptual structures into disjoint classes of structures called constraint, definitional, constructional, link and collection of concepts, respectively.

Figure 13 shows the product primitive instances that identify the criteria that must be met by a good quality product. There are four kinds of fixed structures as shown. Again, these are created by the drag and drop facilities of the generic-CAME tool.

Operations identify the instances of *process* primitives of the generic model that operate upon conceptual and fixed structures to provide product manipulation and quality checking capability to application engineers. Again, the generic-CAME tool is used to produce Fig. 14.

There are two operations in the basic life cycle class, create and delete. The relational class of operations is axiomatically defined and this definition can be found in [15].

A purpose is an instance of the method block of the generic model. It is a pair

$$\text{Purpose} = <\text{structure, operation}> .$$

Thus, the following are purposes

$$<\text{simple constructional structure, create}>$$
$$<\text{simple constructional structure, method constraint,}$$
$$\text{method constraint enforcement}> . \quad (30)$$

The definition of the set of meaningful purposes of the meta-model is supported by the generic-CAME tool which generates the cross product of structure and operation. This cross product is presented and the meaningful purposes are then selected. The complete set of purposes can be found in [25].

Finally, the set of dependencies of different types are instantiated and the generic-CAME tool keeps an internal representation of the dependency graphs.

The enactment algorithm is inherited by the meta-model. Therefore, its full enactment power including guidance, tracing and backtracking is available. The meta-model is validated using this algorithm to enact typical purposes expected to be found in a method which, in turn, shall be produced from the meta-model.

Appendix 2: Glossary

Product primitive	The smallest unit that can be operated upon in a method
Process primitive	An atomic action that can be enacted
Method primitive	The smallest meaningful process unit of a method that can be enacted. It is a pair < product primitive, process primitive >
Complex method block	A composition of simpler method blocks
Abstract method block	A generalization of a method block
Method block	It can be either a method primitive, a complex or an abstract method block
Product part	A part of a product that is the focus of further development
DPC	The product part together with the method block that can operate upon it. It is the smallest unit of the development process
Development process	The sequence of DPCs enacted to build the product

References

1. Hornby AS (2000) Oxford advanced learner's dictionary of current English. Oxford University press, Oxford
2. Brinkkemper S, Saeki M, Harmsen F (1999) Meta-modelling based assembly techniques for situational method engineering. Inform Syst 24(3):209–228
3. Plihon V, Rolland C (1995) Modelling ways-of-working. In: Proceedings of CAiSE'95, Springer, Berlin Heidelberg New York
4. Ralyte J, Rolland C (2001) An assembly process model for method engineering, In: Proceedings of CAiSE'01, Springer, Berlin Heidelberg New York
5. Rolland C, Prakash N, Benjamen A (1999) A multi-model view of process modelling. REJ 4(4):169–187
6. Prakash N (1994) A process view of methodologies, advanced information systems engineering. In: Wijers, Brinkkemper, Wasserman (eds) LNCS 811, Springer, Berlin Heidelberg New York, pp 339–352
7. Smolander K (1991) OPRR: a model for modeling systems development methods, next generation of CASE tools. IOS Press, Amsterdam
8. Souveyet C (1991) Validation des Specifications Conceptuelles d'un Systeme d'information, Ph.D. thesis, University of Paris
9. Brinkkemper S (1990) Formalisation of information systems modelling, Ph.D. thesis, University of Nijmegen, Thesis Publishers, Amsterdam

10. Wijers GM (1991) Modelling support in information systems development, Ph.D. thesis, University of Delft, Thesis Publishers, Amsterdam
11. Grosz G et al (1997) Modelling and engineering the requirements engineering process: an overview of the NATURE approach. REJ 2(3):115–131
12. Prakash N (1999) On method statics and dynamics. Inform Syst J 24(8):613–637
13. Jarke M et al (1994) Experience based method evaluation and improvement: a process modelling approach. In: Verrijn-Stuart, Olle (eds) Methods and associated tools for the information systems life cycle, Elsevier, Amsterdam, pp 1–28
14. Harmsen F et al (1994) Situational method engineering for information system project approaches. In: Verrijn-Stuart, Olle (eds) Methods and associated tools for the information systems life cycle, Elsevier, Amsterdam, pp 169–194
15. Prakash N (1997) Towards a formal definition of methods. REJ 2(1):23–50
16. Prakash N, Bhatia MPS (2002) Generic models for engineering methods of diverse domains, advanced information systems engineering (CAiSE 02). In: Pidduck, Mylopoulos, Woo, Ozsu (eds), LNCS 2348:612–625
17. Checkland OP, Scholes J (1990) Soft systems methodology in action. Wiley, Chichester
18. Atkinson CJ (1997) Soft information systems and technologies methodology (SISTeM): a case study on developing the electronic patient record. REJ 1–22
19. Blum BI (1994) A taxonomy of software development methods. CACM, pp 82–94
20. Davis AM et al (1988) A strategy for comparing alternative software development life cycle models. IEEETSE, pp 1453–1461
21. Lyttinen K (1987) A taxonomic perspective of information systems development: theoretical constructs and recommendations. In: Boland, Herschheim (eds) Critical issues in information systems research, Wiley, New York, pp 3–41
22. Krogstie J, Solvberg A (1996) A classification of methodological framework for computerized information systems support in organizations. In: Brinkkemper, Lyytinen, Welke (eds) Method engineering: principles of method construction and tool support, Chapman & Hall, London, pp 278–295
23. Rolland C, Souveyet C, Ben Achour C (1998) Guiding goal modelling and scenarios. IEEETSE 24(12):1055–1071
24. Prakash N, Sibal R (1999) Modelling method heuristics for better quality products, advanced information systems engineering. In: Jarke M, Oberweis A, (eds) LNCS 1626, Springer, Berlin Heidelberg New York, pp 429–433
25. Prakash N, Bhatia MPS (2003) Developing application centric methods. In: Eder, Missikoff (eds) CAiSE Forum, pp 225–228
26. Ralyte J et al (2003) Towards a generic model for situational method engineering, advanced information systems engineering. In: Eder J, Missikoff M (eds) CAiSE 2003, LNCS 2681, Springer, Berlin Heidelberg New York, pp 95–110