Design and Implementation of a Method Base Management System for a Situational CASE Environment

Frank Harmsen, Sjaak Brinkkemper
University of Twente, Department of Computer Science,
IS Design Methodology Group,
P.O. Box 217, 7500 AE Enschede, the Netherlands,
tel. +31.53.893690 / fax +31.53.339605
E-mail {harmsen,brinkkemper}@cs.utwente.nl

Abstract

Situational Method Engineering focuses on configuration of system development methods (SDMs) tuned to the situation of a project at hand. Situational methods are assembled from parts of existing SDMs, so-called method fragments, that are selected to match the project situation. The complex task of selecting appropriate method fragments and assembling them into a method requires effective automated support. This paper describes the architecture of a tool prototype offering such support. We present the structure of its central repository, a Method Base containing method fragments. The functions to store, select and assemble these method fragments are offered by a stratified Method Base Management System tool component, which is described as well.

1 Introduction

Current systems development methods are situation independent, and claim to be applicable in virtually any application domain. However, due to the ever increasing complexity and diversity of information systems, software development projects do not require general purpose systems development and management methods, but specialised, dedicated approaches [1;2;3;4]. These so-called situational methods should take at least the *situation* in which they are applied into account, for instance the system type, the DBMS platform, the experience of the project members, etc., thus obtaining a better fit between approach to be followed and the required tasks and deliverables.

Situational Method Engineering [5] is the research area having the philosophy, that system development projects should strive for *controlled flexibility*, being the balance between rigid general-purpose methods and ad-hoc, flexible development. To engineer a project-specific or *situational* method, first a characterisation of the situation in which the method will be applied, often a project, is made. This characterisation is input to the selection process, where parts of existing system development methods are retrieved.

Such parts, called *method fragments*, stored in a *Method Base*, address both the method process perspective, such as modelling steps and stages, and the method product perspective, such as descriptions of reports, models, and diagrams. Method fragments addressing the process perspective are called process fragments, fragments addressing the product perspective are called product fragments. The unrelated method fragments are then assembled into a situational method, using a large number of assembly rules [6] to ensure internal consistency and completeness. Finally, the situational method is forwarded to the systems developers in the project. As the project may not be definitely clear at the start, a further elaboration of the situational method can be performed during the project. Similarly, drastic changes in the project require to change the situational method by the replacement of inappropriate fragments.

The complexity of this process requires support by a CAME (Computer Aided Method Engineering) tool [1]. Several authors have acknowledged the need for computerised support for Method Engineering. One of the academic tool prototypes is the *Method Base* tool [7]. The database associated with this tool contains information about documents to be produced and activities to be performed. Method Base enables the software engineer to select a method that fits the project at hand, is able to guide and navigate the user through the method, and allows for multi-view representation. The tool does not specifically aim at assembling situational methods, but offers facilities for method customisation. Hidding et al. have proposed the *Solution Configuration Tool* (SCT), from which parts of a comprehensive method can be retrieved [8]. SCT should be regarded as an on-line knowledge base enabling a methodology department to accumulate experience about method application. The *Method Engineering Tool*, offering facilities to construct diagram editors, textual reports and repositories [9],

0-8186-7171-8/95 $04.00

supports method specification. It consists of a so-called Meta Repository, containing all the necessary specifications and rules to generate a CASE workbench. *MetaEdit* [10] provides facilities to specify and construct single diagram editors, for instance a DFD editor, and an associated repository in a simple way.

Besides academic prototypes, some commercial tools are available, most notably Ernst & Young's *Navigator* and James Martin & Co.'s *Architect*. Navigator consists of an "Automated Methods Environment", which enables the project manager to select descriptions of activities and deliverables according to a project description, to link company standards with product descriptions, and to combine all these into a project plan. The project plan can contain links to tools, such as a word processor or a CASE tool's diagram editor. Architect is similar to Navigator, but provides more facilities to modify the descriptions of activities and products. It contains less support for project characterisation. Both tools are based on the Information Engineering method [11].

None of these Method Engineering support tools offers an integrated set of functions facilitating Situational Method Engineering. A CAME tool should support the uniform and high-level specification of system development methods, and should allow for the construction of systems development tools. It should support the *method engineer* in selecting appropriate method fragments, in assembling the method fragments, and in transfering the resulting situational method to the systems development project.

The requirements mentioned above were the motivation for developing a CAME tool, which we called Decamerone. This paper focuses on the design and implementation of the kernel of Decamerone, its Method Base and associated Method Base Management System. The paper is organised as follows. In section two, the overall technical architecture of Decamerone is presented, whereas section three deals with the design of the Method Base structure. Section four focuses on the Method Base Management System, which offers Method Base access and modification functions. The paper ends with conclusions and suggestions for further research.

2 A Situational CASE Environment

To support Situational Method Engineering, a CAME tool should provide at least the following functionality:

- *Representation*, allowing for the description of method fragments. Method fragments can be described in any kind of language that is able to represent products or processes. Saeki and Wen-Yin [12], for instance, are using Object Z to represent method fragments.
- *Administration*, which provides facilities to insert and modify method fragments in the Method Base. For instance, it is possible that a process fragment needs to be extended by an additional sub-activity. The adaptation needed is performed by an administration function.
- *Selection*, which provides functions to retrieve method fragments from the Method Base. The method engineer should be able to execute queries on the Method Base, such as: "Select all method fragments that have to do with object oriented programming requiring average experience from the programmers".
- *Assembly*, allowing for the assembly of selected method fragments. Process fragments and product fragments can be combined to form larger components, eventually leading to a situational method.

Figure 2.1 depicts the architecture of Decamerone. Decamerone is implemented in the meta-CASE environment Maestro II [13], thus simplifying many implementation issues. The situational CASE environment consists of two main components: a CAME component, and a CASE component. The CAME component is built upon the Method Base Management System (see section 4) and provides facilities for specifying, storing, and selecting method fragments, and for assembling method fragments into a situational method. The CASE component uses the situational method as a definition for its repository structure, its editors and report generators, and its process engine.

In this paper, we will mainly deal with the CAME component, which will be explained in more detail in the next sub-sections.

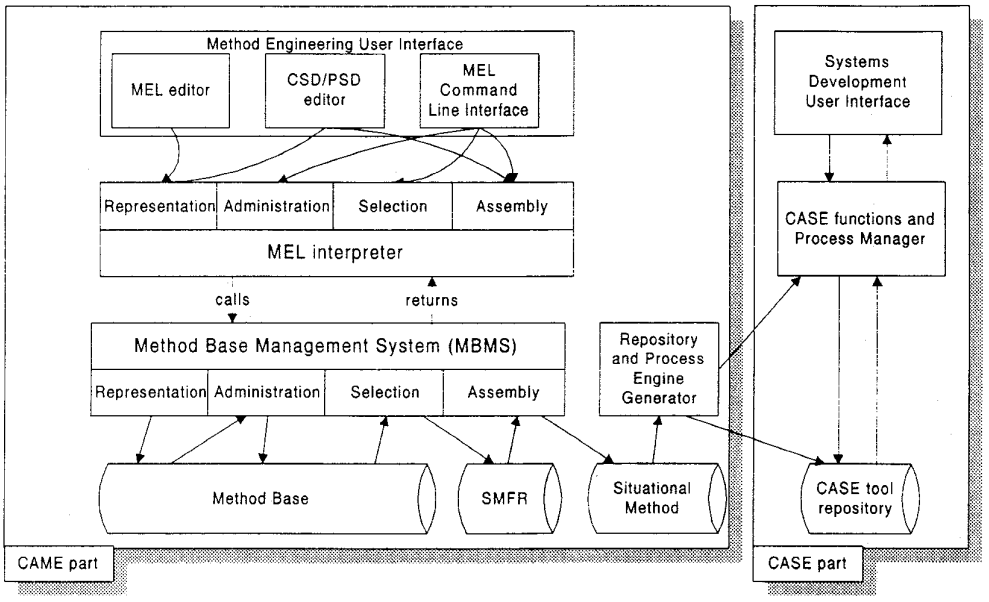

Figure 2.1 *Architecture of Decamerone*

PROCESS Create Prototype business process:
ID METH1/A2.1.2.1;
LAYER Diagram;
SOURCE Method/1; # used with permission of Andersen Consulting #
PARENT Describe Requirements;
TYPE Abstraction;
REQUIRED {Entity-relationship Diagram, Organisation-process Diagram, Workflow Diagram, User Requirements};
(
- Define Prototype scope;
- Develop Prototype review plan;
- Select Prototype environment;
REPEAT
  - Design Prototype;
  - Build Prototype;
  - Test Prototype;
  - Review Prototype;
  - {Document Reactions | Document Revisions requested}

UNTIL Prototype accepted
)
DELIVERABLES {Business process Prototype, Issue List}.

Figure 2.2 ***MEL** process fragment specification*

2.1 The MEL interpreter

The Method Engineering Language (**MEL**) provides concepts and constructs dedicated to describing, selecting, and manipulating method fragments [14]. **MEL** is also used for other purposes, for instance CASE tool repository specification and integration [15]. Examples of a **MEL** process fragment specification and a **MEL** assembly operation are depicted in figures 2.2 and 2.3 respectively.

The specification depicted in figure 2.2 describes an activity that has a number of *properties*, such as its granularity layer (see section three) and the systems development method from which it was taken ("Source"). The products that the process fragment *requires* as input, and the products that it *delivers* are specified as well. Between the brackets, the process flow is indicated. Note that a bar between process names indicates parallelism.

Join Technical data Model **With** {Module Structure, User-interface Design, System-interface Design} **Into** Technical design Report.

Figure 2.3 ***MEL** assembly operation*

The assembly operation shown combines several product descriptions, and calls the resulting product a "Technical design Report". Assembly rules are used by

the interpreter to assure a consistent situational method. An example of such a rule is shown in figure 3.1.

The **MEL** interpreter translates specifications and operations into corresponding Method Base Management System function sequences. An example of such a translation will be shown in section four.

2.2 The User Interface

The **MEL** command line interface offers facilities to modify, select, and combine method fragments in a high-level method engineering language. Besides the possibility to textually specify and assemble method fragments, we have also incorporated a Concept Structure Diagram (CSD) editor for specification and assembly of product fragments, and a Process Structure Diagram (PSD) editor for specification and assembly of process fragments [5]. CSD is a dialect of Entity-Relationship diagrams, extended with complex objects to allow for specification of composite models and diagrams. PSD offers the concepts "Task", "Trigger", and "Product", the latter being a composite CSD object. A menu structure provides links to the various tool components. It contains, in addition to that, dialogue boxes to perform queries, and to retrieve statistical information about the Method Base.

2.3 Decamerone databases

Decamerone uses four databases: the Method Base, the Selected Method Fragments Repository (SMFR), the Situational Method Database, and the CASE tool repository. The *Method Base* is the central repository of Decamerone, containing method fragments and their relationships. The *SMFR* contains the unconnected method fragments that have been chosen for incorporation into a situational method. The *Situational Method Database* contains the assembled situational method. The data perspective of the situational method, containing, for instance, the concept "Data store", the association "Data Store is described by Entity", or the description (*meta data model*) of a Data Flow Diagram, describe the structure of the *CASE tool repository*, which will be used during the project to store all kinds of products.

2.4 Generators

For practical use, the situational method has to be processed by a repository structure generator and a process engine generator. As Decamerone is completely implemented in Maestro II, generating a repository structure involves the conversion of the situational method data perspective (a set of objects) into a Maestro II database structure (a set of object classes). The process engine generator makes use of the Maestro II facilities to define process managers, which can force the CASE tool user to invoke for instance certain diagram editors.

3 Design of the Method Base structure

3.1 Classification of method fragments

The method fragments in the Method Base are classified using the dimensions *perspective*, *abstraction level*, and *granularity layer*.

The perspective dimension considers the *product* perspective and the *process* perspective on methods. Product fragments represent deliverables, milestone documents, models, diagrams, etc. They can consist of other product fragments. Process fragments represent the stages, activities and tasks to be carried out. Process fragments have precedence relationships with each other, can consist of other process fragments, and require and produce product fragments.

The abstraction level dimension constitutes of the *external level*, the *conceptual level*, and the *technical level*. Conceptual method fragments are descriptions of systems development methods or part thereof, for instance a description of an Entity-relationship diagram, or JSD's Network phase. External method fragments [14] are introduced to accommodate multiple views of project roles on methods (cf. [16;17]). Views to be distinguished include the project manager's view, the analyst's view, and the programmer's view. Technical method fragments are the operational parts of a method, i.e. tools and repositories. External method fragments are *derived* from conceptual method fragments, which are *supported* by technical method fragments.

The granularity layer dimension is introduced to deal with method fragments of different levels of decomposition. When assembling a situational method, the decomposition levels of the various method fragments, taken from different SDMs, have to be aligned with each other. However, different SDMs have different numbers of decomposition levels, which makes alignment cumbersome. Therefore we have introduced granularity layers. A granularity layer of a particular method fragment is completely determined by its properties, and not by its level in the decomposition tree. For systems development methods, five granularity layers can be considered:

- **Method layer** The highest layer addresses the entire method, and is therefore called the Method layer. An example of such a method fragment is a method like JSD, or Information Engineering.
- **Stage layer** This layer consists of the direct components of a method, producing milestone deliverables. Such method fragments, for instance a Requirements Analysis stage, address different abstraction levels of the object system.
- **Model layer** Although *model* is a generic term, systems developers use it to indicate products describing a specific aspect of the object system, for instance data or processes. Models, and the activities to create them, belong to the Model layer. An example

of a method fragment on the model layer is "Create test suite", or "Technical data model".

- **Diagram layer** Models can be represented in many different ways and at various levels of detail, and consist therefore of components, often diagrams like DFD's, Action Diagrams or Decomposition Diagrams. These model components, and the activities to produce them, reside on the Diagram layer.
- **Concept layer** Method fragments residing on the Concept layer are the concepts and associations describing the diagram components, as well as the manipulations with these concepts and relationships. Examples of such method fragments are "Entity", "Actor" and "Determine actors".

Note, that granularity layer is not a relative notion within one method. However, within one granularity layer, levelling is allowed, in order to enable representation of methods consisting of more than five decomposition levels.

3.2 Properties of method fragments

We distinguish variable project properties and intrinsic properties. Intrinsic properties receive a value in the Method Base, whereas project properties receive their respective values during the system development project. The first category includes:

- LAYER, the granularity layer of the method fragment, which is of property type GRAN_LAYER = {Method, Stage, Model, Diagram, Concept}.
- GOAL, the goal to be achieved with the method fragment, which is of type FRAG_GOAL = $\wp$({{a,b,c} | a ∈ "Verbs", b ∈ "NounProperties", c ∈ "Nouns"}).
- SOURCE, the name of the SDM from which the method fragment is taken, which is a string of characters.
- EXPERIENCE, the amount of experience needed by a project member to perform or apply a method fragment. The associated property type is EXP_LEVEL = {Little, Average, Much, Very much}.
- TRAINING, the amount of person days required to train a project member in performing a process fragment.
- TYPE describes whether a process fragment is an abstraction step, a form conversion step, a decision, a review step, or a checking step [18].
- RESPONSIBLE, the actor type responsible for -the instance of- the method fragment. Property type is ACTOR = {Commissioning Agent, User, Project manager, Analyst, Functional Designer, Technical Designer, Programmer, System Tester, Acceptance Tester, Database Administrator},
- EXECUTOR, the actor type that executes the process fragment, also of type ACTOR.
- CREATOR, the actor type or set of actor types in the systems development project creating instances of the product fragment, which is of type $\wp$(ACTOR).
- FOR, the actor type or set of actor types for whom -the instance of- the product fragment is made, also of type $\wp$(ACTOR).

Examples of first order properties are well-known method fragment instance attributes like creation date, comments, definition, and instances of actor types responsible for, performing, creating, or receiving the method fragment instance. An important first order property is the experience record associated with each method fragment. In this record, project member can enter their experiences with a method fragment. Dependent on the contents of these experience records, a method engineer can modify a method fragment accordingly.

3.3 Formalisation of the Method Base structure

The global structure of the Method Base is given by a first order predicate logic specification, which constitutes the basis for the specification of rules, as will shown at the end of this section.

We define: $M = R \cup P$, the set of method fragments, where R represents the set of product fragments, and P the set of process fragments. Note that $R \cap P = \varnothing$.

The following predicates are used to express relationships between method fragments:

predicate consists of **over** $(R \times R) \cup (P \times P)$, indicating the existence of a "consists of" relationship between product fragments and between process fragments,

predicate produces **over** $P \times R$, which holds if a process fragment produces a product fragment,

predicate requires **over** $P \times R$, which holds if a process fragment requires a product fragment,

predicate precedes **over** $P \times P$, indicating the existence of a precedence relationship between product fragments. We assume that precedence is transitive,

predicate is supported by **over** $(R \times R) \cup (P \times P)$, which holds if a conceptual method fragment is supported by a technical method fragment,

predicate is view upon **over** $(R \times R) \cup (P \times P)$, indicating that an external method fragment is a view upon a conceptual method fragment.

Furthermore, for each property a function is defined, for instance:

function type **over** P **to** PROCESS_TYPE, yielding the *type* of process,

function layer **over** M **to** LAYER, yielding the granularity layer,

function goal **over** M **to** GOAL, yielding the goal of a method fragment.

Sets, predicates, and functions are used to formally

define *method assembly rules*, which will be incorporated in Decamerone to support consistent combination of method fragments. One example of a method assembly rule, taken from a set of 66 rules described in [6], is depicted in figure 3.1.

Process completeness rule
Informal description: all (parts of) product fragments have to be output from a process fragment. All method, model and diagram layer product fragments have to be produced by process fragments; Stage layer product fragments do not have to be produced themselves, but their contents on lower layers do: e.g. "Business Re-engineering Master plan" from the method Method/1 is not created itself, but its contents are. Concept layer product fragments (concepts) do not have to be produced themselves, if their parent is produced by a non-decomposable process fragment.

Formal specification:

$\forall r_1 \in R\ \exists p_1 \in P\ \forall r_2 \in R\ \exists p_2 \in P\ \forall p_3 \in P$ [

$(\text{layer}(r_1) \in \{\text{Diagram,Model,Method}\} \rightarrow \text{produces}(p_1,r_1)) \wedge$

$(\text{layer}(r_1) = \text{Stage} \rightarrow (\text{produces}(p_1,r_1) \vee ((\text{consists of}(r_1,r_2) \wedge$

$\text{layer}(r_2) = \text{Diagram}) \rightarrow \text{produces}(p_2,r_2)\ ))) \wedge$

$(\text{layer}(r_1) = \text{Concept} \rightarrow (\text{produces}(p_1,r_1) \vee (\text{consists of}(r_2,r_1) \wedge$

$\text{produces}(p_2,r_2) \wedge \neg\text{consists of}(p_2,p_3))\ )\ ]$

Figure 3.1 Example of a method assembly rule

4 Implementation of the Method Base Management System

The Method Base Management System (MBMS) is the kernel of Decamerone and provides the operations necessary to interact with the Method Base. As was shown in figure 2.1, the MBMS is called by, and returns values to, the **MEL** interpreter. It is developed as an application within the Maestro II meta-CASE environment.

4.1 Overview of Maestro II

Maestro II is a meta-CASE environment developed and marketed by the German Softlab company. Since the tool can be customised to a very large extent, it can be used to develop project-specific CASE tools. Due to the comprehensiveness of Maestro II, only the most relevant parts are listed here. These parts are:

- The *Object Management System* (OMS), a multi-user on-line repository, represents the data base management system of Maestro II. All data concerning both application and method development are stored in databases of the OMS. The OMS stores data as object classes and objects.
- *Project and Configuration Management System* (PCMS), dealing with specification and execution of process managers. PCMS is the Maestro II counterpart of an SDM's process flow, supporting its user in performing system development activities and invoking the appropriate tools. The PCMS is also used to define deliverables, and to define and maintain relationships between activities and deliverables.
- The Maestro II environment can be customised by writing procedures in the systems programming language *Prolan*. This proprietary language, similar to C and assembler, provides means to build a user interface, and offers function calls to the OMS as well.
- Dedicated *editors*, both graphical and textual, which are not only used for entering ordinary text files and diagrams, but also to specify and customise diagram editor descriptions, Prolan programs, deliverable templates, and OMS database structures. Relationships exist between on the one hand the steps and deliverables defined with the PCMS, and on the other hand diagram- and text editors.

Implementing a CAME environment using such a meta-CASE environment offers advantages over using a conventional programming language. Maestro II offers the availability of symbol editors, a CASE tool oriented DBMS, and reporting and text/graphics editing facilities [13]. A limit of this approach is, that the resulting CAME tool is not a stand-alone application, but should always be used in conjunction with Maestro II.

4.2 Implementation of the Method Base Management System in Maestro II

The Method Base Management System, completely programmed in Prolan, is partitioned into the following layers:

1. Tool-specific layer, which provides a set of atomic operations defined in terms of OMS calls. This layer serves as an interface between Maestro II's OMS and the rest of the MBMS.
2. Basic CAME functions layer, which assures the availability of all necessary basic operations such as creating or modifying a method fragment, a relationship, or an attribute.
3. Compound CAME functions layer, which provides aggregated functions to perform more complex operations, such as creating a product fragment with its properties.

The three layers have been introduced to abstract from different types of problems and solve them per type in modules. The result is a structured realisation of the MBMS that allows top-down and bottom-up iterations without loosing the overview over the different components and their functions.

The function and implementation of the MBMS will be illustrated with the following example. Suppose the method engineer has specified the following product

fragment with the **MEL** editor, which is to be stored in the Method Base:

```
PRODUCT User-interface Standard:
 ID METH1/P4.3.10;
 LAYER Model;
 GOAL {Description Screen Layout, Description System
       Usage, Description User-interface Response};
 SOURCE Method/1; # used with permission of Andersen
       Consulting #
 PART OF Business process Design;
 CREATED BY Complete Dialog design;
 (
       - Screen model List;
       - Window model List;
       - Widget model List;
       - Keyboard usage Instructions;
       - Mouse usage Instructions;
       - Feedback Description;
       - Error prevention Description;
       - Multiple language support Description
 ).
```

Figure 4.1 Specification of a product fragment

The **MEL** interpreter first assigns attribute values to global variables, as Prolan is incapable of having more than one parameter passed to procedures. For each predicate and function defined in the Method Base structure specification, one or more global variables have been declared in the MBMS. The alternative to this fairly awkward solution, passing complex structures, is only used where time-criticality is no issue. In figure 4.2, some assignments are shown for our example.

```
sysvar("Name") = "User-interface
                 Standard";
sysvar("Perspect") = "Product";
sysvar("Abstract") = "Conceptual";
sysvar("Layer") = "Model";
sysvar("Created") = "Complete Dialog
                 design";
```

Figure 4.2 Assignment of global variables

Each layer uses the information contained in the global variables to create objects in the Method Base. Next, the MBMS Compound CAME layer procedure *CR_Fragment*, with parameter METH1/P4.3.10, is invoked, which first inspects whether the fragment is legal according to the attribute constraints. When the constraints are met, a method fragment object is created in the Method Base. When *CR_Fragment* fails, it will remove the method fragment to keep the Method Base consistent.

CR_Fragment invokes the Basic CAME layer procedures *Create_Fragment* and *AttrRestrictions*. *Create_Fragment* takes care of the creation of the method fragment object, as well as the product fragment object including the corresponding sub/super-type relation. To simplify retrieval, each fragment is stored both as a method fragment object and as a product- or process fragment object. *AttrRestrictions* checks, whether the object's attributes meet the defined constraints. For instance, the following piece of Prolan code checks, whether the granularity layer has a valid value:

```
Layer = SYSVAR('LAYER')
   [SYSVAR('RETURN_CODE')EQUAL Space?
   [ Layer LESS_THEN MethodLayer?
   | Layer GREATER_THEN ConceptLayer?]?
     SYSVAR('RETURN_CODE') = 'Level not in
       range '
   ]
```

Figure 4.3 Check of an attribute constraint in Prolan

After creation of the product fragment and check of the attribute constraints, the *CR_MethodAttr* procedure creates the attributes and their values, such as fragment name="User-interface standard", layer="Model", source="Method/1", etc. In the source code depicted in figure 4.4, part of *CR_MethodAttr*, the tool-specific procedure *CreateAttribute* is called to actually store the attribute and its value in the Method Base.

The result of the Compound CAME layer and Basic CAME procedures is, that all information regarding the fragment is broken down into units that can be stored in an OMS database. The Method Base is implemented as an OMS database, which is started and accessed by means of Prolan procedure calls offered by the Tool-specific procedures. The *CreateObject* and *CreateRelation* procedures create a CAME object in the Object Management System, as well as a relation between two CAME objects, respectively. The attributes are stored in the OMS by the *CreateAttribute* procedure. Part of the *CreateAttribute* source code is depicted in figure 4.5.

After having seen the low-level nature of actually accessing an OMS database, the advantages of having an MBMS as a CAME tool kernel are obvious: the functionality of the tool is easier to adapt, the structure of the Method Base is easier to modify, programming the system requires less knowledge of a proprietary language like Prolan, and the CAME tool obtains a higher degree of portability.

```
Operation.Process                  = AttrCall.Operation
     Operation.Object                   = SYSVAR('FRAGID')
     Operation.ActiveBase               = SYSVAR('ACTIVE_BASE')
     FrgSource                          = SYSVAR("Source"')

     Operation.ObjectClass              = MethodFragClassName

     ;.............. Create SourceAttribute ..................
     [ Operation.ReturnCode EQUAL Attribute_Ok ?
          AttributeOperation (CreateAttribute, FrgSource, MethodFragSource)
     ]
```

Figure 4.4 Creation of an attribute and its value in the Method Base

```
$HDR: CreateAttribute                (Operation OPERAT_TYPE)

$DATA:
    oms_par              OMS_PARAM
    oms_lst              OMS_LIST
    oms_val              OMS_VALUE(OMS_VALUE_LEN)

$BEGIN:
     ;---------------------- PARAMETERS --------------------
     oms_par.OBJ_CL         = Operation.ObjectClass
     oms_par.OBJ            = Operation.Object
     oms_par.ATT_CL         = Operation.AttributeClass
     oms_val.VALUE          = Operation.Attribute&Delimitor
     ;-----------------------------------------------------
     oms_par.OPCODE         = InsertObjectAttribute

     ;....................... database ......................
     START_OMS(oms_par, Operation.ActiveBase)

     OMS_INIT_VALUE(oms_val, Operation.AttributeClass)
     OMS_SET_PARAM(oms_par, oms_val)

     CALL_OMS(oms_par, Operation.ActiveBase)
     ;---------------------- PARAMETERS --------------------
     Operation.ReturnCode    = oms_par.RC
     ;-----------------------------------------------------
     .
$END
```

Figure 4.5 Creating an attribute by performing an OMS call

5 Conclusions and further research

The Method Base Management System, implemented in the meta-CASE environment Maestro II, is the kernel of the Decamerone* situational tool environment that assists in efficiently and effectively configuring and applying project-specific systems development methods. We have shown an overall architecture of Decamerone, as well as a more detailed treatment of the Method Base and the Method Base Management System.

The implementation of the **MEL** editor and interpreter is currently addressed in the Pampinea project. Further future research concerning Decamerone focuses on the incorporation of quality enforcing rules, currently specified in predicate logic (Filomena project), the implementation of support for method assembly by the Methodology Data Model (Filostrate project), and the implementation of repository and process engine generators (Neifile project).

* "Il Decamerone", by the Italian writer Giovanni Boccaccio (1313-1375), contains 100 stories, told on 10 days by 10 people who were on the run for the plague in Florence. The names of the projects mentioned in the conclusions are the names of the 10 persons. Il Decamerone can be regarded as a "story base", rather than a Method Base.