Computer-Aided Method Engineering: An Analysis of Existing Environments

Ali Niknafs and Raman Ramsin

Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
niknafs@ce.sharif.edu, ramsin@sharif.edu

Abstract. Analogous to Computer-Aided Software Engineering (CASE), which aims to facilitate Software Engineering through specialized tools, Computer-Aided Method Engineering (CAME) strives to support a wide range of activities carried out by method engineers. Although there is consensus on the importance of tool support in method engineering, existing CAME environments are incomplete prototypes, each covering just a few steps of the method engineering process. This paper summarizes the history and the state of the practice in CAME technology, and provides criteria-based critique on existing CAME environments, thus highlighting their strengths and weaknesses.

Keywords: Software Development Methodologies, Method Engineering, Computer-Aided Method Engineering, Criteria-Based Analysis.

1 Introduction

"If it says one size fits all, it doesn't fit anyone": Although it is safe to assume that every methodology fits at least one project situation, this variant of the Murphy's Law stresses the fact that there is no general-purpose methodology applicable to all different situations. This motivates the development of project-specific methodologies, using an approach known as Situational Method Engineering (SME) [1], a complex and error-prone process that cannot be properly performed without automated support. The automated support required is provided by Computer-Aided Method Engineering (CAME) environments [1, 2, 3, 4, 5, 6]. A CAME environment is composed of a set of correlated tools aiming to facilitate, in its ideal form, the entire SME process. CAME technology dates back to the early days of method engineering, when several academic prototypes were first introduced.

A method is composed of two parts: The *product part* which captures the product-related knowledge, and the *process part* encompassing the activity-related aspects of the method. Due to this division, two types of method fragments can be defined: *Product* fragments are artifacts such as models, diagrams, and documents, whereas activities, stages, and tasks are considered *Process* fragments. To enable computerized support for SME, method fragments need to be stored in a repository called the Method Base. They thus need to be described in a formal way. Several method representation languages have been proposed for this purpose, which are either textual or graphical, or both. Graphical languages are called *meta-modeling languages*, e.g. GOPPRR in MetaEdit+ [7]. Object Z [8] is a textual language, whereas the Method

Engineering Language (MEL) [9] is both textual and graphical. Meta-modeling languages are more popular than textual ones. This popularity is mainly because they are easier to use, learn, and implement.

As shown in Fig. 1, CAME environments are made up of two parts: The CAME part provides facilities for method engineering, whereas the CASE part offers means for the generation of CASE tools and process support environments. The set of Method Engineering Tools and the Method Base form the main elements of the CAME part. The Method Engineering Toolset offers tools for facilitating the work of method engineers, e.g. for extracting components of existing methods and storing them in the Method Base. The Method Base upon which a CAME environment is built is the kernel of the CAME environment. The method obtained from the CAME part will be fed as input to the CASE part. The CASE Generator gets the product part of methods and generates the project specific CASE tool. Process-centered Software Engineering Environments (PSEEs) are used for generating process support environments based on the process part of methods.

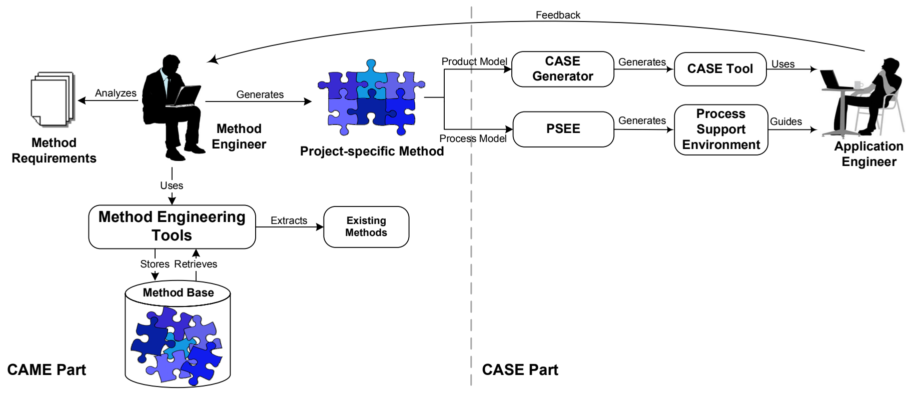

Fig. 1. General architecture of CAME environments

In [10], three distinct approaches to SME are proposed. The *assembly-based* approach is the most common and consists of three steps: specifying method requirements, selecting method fragments, and assembling them into a method. *Extension-based* SME aims at adapting and extending an existing method with new features; whereas in the *paradigm-based* approach a new method is developed by instantiating, abstracting or adapting an existing meta-model. As we will see in the following sections, almost all the existing CAME environments support the assembly-based approach, and other approaches are almost completely overlooked.

CAME environments can be classified as *product-oriented* and *process-oriented*, depending on the way they facilitate the enactment of the method engineering process. Those that focus on modeling the product-related issues of methods, and provide less support for the method's process model and its enactment, are classified as product-oriented CAME environments. Process-oriented CAME environments deal with the process-related issues of methods and support the enactment of the process model.

Most of the existing CAME environments fit into the first class, due to their emphasis on modeling the product part of methods.

Coverage of the method engineering process is one of the major shortcomings of existing CAME environments. Several method engineering processes have been proposed in the literature [10, 11, 12, 13], yet they can all be considered as consisting of the following generic phases:

1. *Method Requirements Analysis (MRA)*: focuses on the identification of important features of the method under construction. Method requirements are those features that are expected to be present in an Information Systems Development (ISD) method, such as traceability to requirements or support for umbrella activities [14]. In the MRA phase, the method requirements should be defined in a formal way, and should therefore precisely describe the features that the desired method needs to offer.
2. *Method Design (MD)*: focuses on determining a blueprint for the method, based on the requirements defined in the previous phase.
3. *Method Implementation (MI)*: focuses on selecting suitable method fragments and assembling them, instantiating an existing meta-model or process pattern, or modifying or extending an existing method. The result of this phase is a set of CASE tools providing means to support the method's product model, and process support environments to guide the application engineer during the ISD project.
4. *Method Test (MT)*: focuses on the verification and validation of the newly developed method. The results of the MRA phase are fed as input to this phase. Testing a newly developed ISD method is similar to testing any other type of system: develop test cases (in this case, sample systems), perform verification and validation, and correct the defects detected. Testing the resulting ISD method is a weak point of existing CAME environments: CAME environments do not offer adequate means for determining whether a newly developed method realizes the predefined method requirements or not.

The aim of this paper is to present a brief overview of past research conducted on CAME environments and the state of the practice as reported and documented by researchers, identifying the shortcomings and thereby offering suggestions for future research. The remainder of the paper is structured as follows: in section 2, several existing CAME environments will be briefly described; these will be analyzed in section 3, based on analysis criteria adapted from the attributes presented in the ISO/IEC 9126 quality model; section 4 contains the conclusions, and outlines our plans for furthering this research.

2 CAME Environments

Several CAME environments have been proposed, yet even though the achievements of these environments have been remarkable, none of them provide a comprehensive set of means for enacting the method engineering process. In this section, we provide concise descriptions for several existing CAME environments, limiting our review to those environments for which adequate documentation is available.

All the environments discussed come from research communities, with very little support from the industry. This is the reason why most of them do not have a long history of usage, even though they enjoy extensive documentation and many years of investigation. There exist tools and environments, such as Rumi [15], that because of little or no available documentation, are very hard to assess and have therefore not been included in this research.

2.1 MERET

The *Methodology Representation Tool (MERET)* is a forerunner of present-day CAME environments, and focuses on method engineering in a product-oriented fashion, dealing with the adaptation and customization of existing methods. MERET presents a comprehensive *methodology representation model* [16] used for the specification of methods. This representation model uses a semantic data-model called ASDM [16] for representing method knowledge. ASDM provides a powerful means for modeling objects and their interrelationships. The root concept used in the methodology representation model is the so-called *MERET object*, which provides the attributes common to all other objects (e.g. name). A MERET object can be a *methodology object* or a *guideline object*: the former consists of all the objects needed for method specification, whereas the latter describes the rules, constraints and experiences relating to the former. Rules are represented in a formal way by means of Horn clauses. Methodology objects are partitioned into *Methods* and *Techniques*, where a method consists of techniques used to develop products. The process model is described by means of *actors, milestones*, *processes* and *deliverables*. A process can be either a *phase* or an *activity*. A technique consists of several *resources*, which are non-human requirements such as any CASE tool feature, and *representation types*, which define the representation of deliverables (e.g. text or a special diagram such as a DFD). MERET provides means for automatic application of consistency checks on the method specification, and the integration of different methods, and the customization of the method produced to specific projects.

2.2 MethodBase

MethodBase is one of the first academic CAME prototypes introduced. The aim of this environment is to facilitate method customization, rather than assembly-based SME. MethodBase assists the method engineer in the selection of a method that best fits the project at hand [1].

The MethodBase system's database consists of complete methods. Therefore, methods can be selected and be customized to fit a project situation. Its data model is divided into product and process parts. The product part consists of the concepts *State*, *Event*, *Data*, *Entity*, and *Association*, whilst the concepts *Activity* and *Activity-relationship*, constitute the process part. By means of the process part, the method engineer can define guidelines to support the enactment of the method process model.

2.3 MetaEdit+

MetaEdit+ is the result of the MetaPHOR project initiated in 1990, and was originally developed as a metaCASE environment. A commercial version of MetaEdit+ has also been released. MetaEdit+ uses techniques similar to those used in assembly-based method engineering. It uses the GOPPRR [7] conceptual data model as its method specification language, which is an evolutionary extension of the OPRR and GOPRR models [17]. The basic constructs of the GOPPRR model are *Graph, Object, Port, Property, Relationship,* and *Role*. Graph is the top-level structure of the meta-model, which is an aggregate concept, composed of objects and their relationships. Object types are the design objects that typically appear as shapes in diagrams. Examples of objects are *Class* in Class Diagrams and *Entity* in Entity Relationship Diagrams. Associations between objects are regarded as their relationships. Each object has a role in the relationships in which it participates. Ports allow additional semantics or constraints on how objects can be connected. Ports can be used as parts of objects, to which roles can be attached. Properties are the characterizing attributes attached to each of the types. MetaEdit+ incorporates a specialized tool for creating and maintaining each of these basic types.

The OPRR meta-modeling language and its extensions – GOPRR and GOPPRR – only deal with the product aspects of methods. However, a process meta-modeling language called GOPRR-p [18, 19] has also been proposed as an extension to GOPRR; this language provides concepts and integration rules for defining different *Process Modeling Languages (PML).*

MetaEdit+ consists of several tool families, among which are *Method Management Tools* [17], aiming at providing CAME functionalities. This tool family consists of the following main parts:

- The *Method Base*: consists of method fragments, symbols needed for representing object types, and generic reports used by the report generator tool to deliver several reports on methods.
- The *Method Assembly System*: consists of tools needed for method assembly, such as *Meta-model Editors*, which provide various tools for specifying the GOPPRR constructs and their connections. The resulting method will be checked for incompleteness and inconsistency by means of the *Consistency Checking System.* The *Symbol Editor* is a drawing tool used for specifying symbols for each object type. A number of reports on the newly developed method can be generated using the report generator tool included in the *Metrics & Statistics System.* Metrics reports are used for analyzing the properties of methods (e.g. the number of objects therein).
- The *Environment Generation System*: is the CASE part of MetaEdit+ offering several generators for delivering a CASE tool, an online help, and a number of reports on the models.

2.4 Decamerone

Decamerone extends the Maestro II metaCASE environment with CAME capabilities, taking advantage of the features already present in Maestro II. Decamerone uses the *Object Management System (OMS)* [1, 21], which is the online object-oriented DBMS of Maestro II. The architecture of Decamerone is shown in Fig 2. As expected, the *Method Base* is the central repository containing method fragments and their relationships. The *Selected Method Fragments Repository (SMFR)* is a subset of the Method Base, containing the selected method fragments for integration into a new method.

The *Situational Method Database* is another subset of the Method Base containing the assembled method. The *CASE tool repository* stores all the products used and produced during the project.

At the core of Decamerone is the *Method Base Management System (MBMS)* [21], which provides facilities for the specification, storage and selection of method fragments and their assembly into a new method. The MBMS is an interface for accessing the OMS databases. This will help the method engineer avoid the low-level complexities of actually accessing an OMS database.

The novel feature of Decamerone is the *Method Engineering Language (MEL)* [9], which is not only used for the representation of method fragments, but also offers constructs for their manipulation operations. Thus, MEL supports the administration, selection and manipulation of fragments. As mentioned before, MEL is both textual and graphical; however, its textual form is much more powerful than the graphical form. Decamerone's user interface consists of three parts: The *MEL Command Line Interface*, which is a text editor used for the selection and manipulation of method fragments in a high-level method engineering language; the graphical editors of the *Concept Structure Diagram (CSD)* and the *Process Structure Diagram (PSD)*, which aid in the specification and assembly of product and process fragments respectively; and the *MEL Editor*, which aids in creating MEL specifications for the graphical forms of the method fragments produced, in order to construct finer grained method fragments.

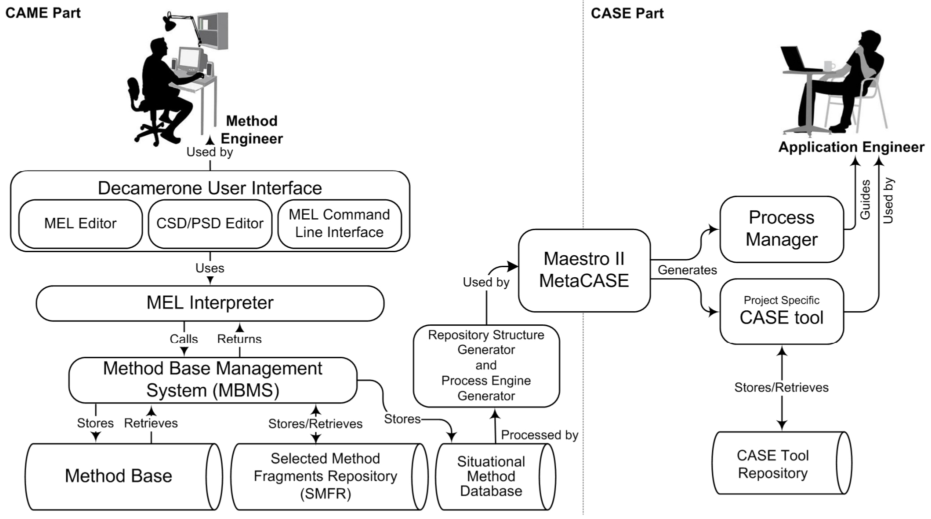

Fig. 2. Architecture of Decamerone

Consistency and completeness of method fragments are checked by the MEL Editor and the CSD/PSD Editors. The *MEL Interpreter* acts as an interface for MBMS and gets user commands and translates them into MBMS function calls. Thus, the MBMS is only called by, and returns values to the MEL interpreter. The newly developed method will be given to Maestro II for CASE tool generation. The *Project and Configuration Management System (PCMS)* is a part of Maestro II used for defining

Process Managers, which enact the method's process model. PCMS offers functionalities for configuration management, project scheduling and estimation. The *Repository Generator* takes the product part of the method as input and generates the CASE tool repository. Notational symbols of the elements manipulated through diagram editors are specified using the *Tool Customizing Interface (TCI).*

Decamerone provides facilities for defining the semantics of method fragments. An *ontology* is defined for product fragments, as well as a *process classification system* [1] for all method fragments, thereby specifying the semantic aspects of the method fragments. The proposed ontology is called the *Methodology Data Model (MDM)* [1] which consists of the basic concepts of ISD products and the associations between them. The process classification system employs the notion of *goal*, which is represented as a tuple *(Action, Measure, Product).* Goals are taken from a process classification, consisting of a set of basic actions in ISD, a set of measures, and a set of product types required in ISD. Basic *actions* are those actions in ISD which have the same effect; a *product* type is a class of products in ISD with the same purpose; and a *measure* is a qualifier of a product, to indicate temporal state, level of detail, or level of abstraction.

2.5 MENTOR

The core component of MENTOR is its *Guidance Engine* [22] which provides guidance to both method engineers and application engineers; MENTOR is therefore a guidance-centered environment. MENTOR uses the NATURE contextual approach to describe method fragments. In the NATURE approach, a method is viewed as a set of method fragments which can either be a *forest*, a *tree* or a *context.* A forest is a set of trees where trees are hierarchies of contexts. Contexts are pairs of the form <*situation*, *decision*>, where *decision* states the intention of the method engineer, and the state of the product that the *decision* can be taken on forms the *situation*. Method fragments, or *method chunks*, are of two kinds: *Components*, which are parts of the product/process model of methods; and generic method construction *Patterns* [23, 24], which can be instantiated to new method fragments. The main components of MENTOR are:

- The Method Engineering Environment, which consists of a set of tools, editors, and browsers for facilitating the work of method engineers [25]. The *product editor* and the *process editor* allow the graphical specification of the product model and the process model respectively. The *method generator* aids in the automatic instantiation of predefined generic patterns stored in the method base. *Browsers* are also provided to help retrieve the necessary method fragments.
- The Application Engineering Environment, which constitutes the CASE part of MENTOR, providing tools for supporting the enactment of methods.
- The Guidance Engine, which advises the method engineer in his method engineering activities and guides the application engineer by executing the resulting process model.
- The Repository, which is organized in three interrelated levels [25]: The *Application Knowledge level*, which is the lower level consisting of the process model and the products under development; the *Method Knowledge level*, which is composed

of method fragments; and the *Method Meta Knowledge level*, which deals with the semantics of the method fragments. Product and process meta-models are placed in the Method Meta Knowledge level, whereas product models and process models are placed into the Method Knowledge level. These two latter levels constitute the Method Base of MENTOR.

2.6 MERU

The main feature of *Method Engineering Using Rules (MERU)* that distinguishes it from other existing CAME environments, is a technical document describing method requirements called the *Method Requirements Specification (MRS)* [11]. MRS is implementation-independent and only expresses the nature of an ISD method. MRS is based on a meta-model called MVM. In MVM, method concepts, which are called *things*, are partitioned into *links*, *constraints* and *product elements*. A *link* is any *thing* of the product that connects two product entities together. *Constraints* are those *things* that can be used by application engineers to specify properties of *links* and *product entities*. Finally, any *thing* that is not a *link* or *constraint* is a *product entity*. The interrelationship between concepts are captured through two relationships: *is composed of*, which identifies concepts that are made up of other concepts; and *is mapped to*, which relates together concepts of two different models. The meta-model proposes to partition things into product entities, constraints and links. The procedure of ME performed by means of MERU consists of 3 main steps. In the first step, called the *Method Requirements Engineering (MRE)* phase, the method engineer expresses the preferred method requirements in the form of an MRS. A language based on the MVM meta-model, called *Method Requirements Specification Language (MRSL)*, is developed to express the MRS. In contrast to other CAME environments, in which a specification language is developed to express the method fragments, MRSL is used only for describing the MRS made by the method engineer. This step is supported by the *MRS Creator*. The MRS thus obtained will then be checked for inconsistencies such as incompleteness and non-conformity with the MVM; this is performed through the *Method Analyzer*. The analysis results are used to provide guidance for refining the MRS. After obtaining the desired MRS, *Method Design* is performed as the next step. Method Design focuses on the translation of the MRS into an instantiation of the MVM. In order to perform this instantiation, for every concept of the given MRS, decisions need to be made as to the type, relationships, and attributes of the concept. The output of this step is called a *plan of instantiation*, which can be modified by the method engineer to obtain the preferred instantiation.

The *Method Construction and Implementation (MCI)* step is then commenced, in which method fragments are created by means of the *Component Builder*. In the method assembly approach used in MERU, method fragments are generated automatically, based on the given MRS. Method fragments are described in terms of *MRS Components (MRSCs)*, which only consider the product part of the methods. The Component Builder uses several predefined rules to identify MRSCs by retrieving the appropriate method fragments from the method base (hence the tool's name). The step is concluded by giving the resulting method description to a metaCASE called RAPID, which generates the appropriate CASE tool.

2.7 Method Editor

Method Editor takes advantage of UML as its meta-modeling technique for expressing the method fragments [3]. Class diagrams are used for the specification of product fragments, while process fragments are described by means of activity diagrams. The process part of a method is attached to each corresponding product fragment, i.e. each product fragment and its development procedure will be shown as a pair of diagrams, a class diagram and an activity diagram.

Method Editor is complemented by a CASE part, so that the Method Editor's output, the resulting ISD method, is fed to the CASE part as input. The CASE part consists of a *Diagram Generator* as the CASE generator, and a *Navigator Generator* which develops a *Navigation Browser* guiding the application engineer through the process of software development. An OCL checker [26] is provided as a part of the resulting CASE tool to check method fragments against the predefined constraints. Any inconsistency seen in the development process will affect the continuation of the whole process, i.e. the process part of methods will be controlled dynamically, forcing adherence to the predefined rules of the method.

The recent version of Method Editor is extended by means of a Version Control System [26], thereby supporting version control and change management of methods or their parts.

3 Analysis of Existing CAME Environments

In this section, we examine the CAME environments introduced in the previous section. Table 1 is a summary of the major features and characteristics of existing CAME environments. In Table 2, the environments have been analyzed and compared with each other based on a few general criteria. In analyzing the environments based on their *Number of Features*, environments with numerous implemented features, partially implemented features, and very few implemented features have been marked as High, Average and Low respectively. The number and importance of the innovations

Table 1. Summary of existing CAME environments

Environment	Coverage of ME Process				SME Approach	Method Representation Language			Process Enactment Support	CASE Tool Generator	Product-Oriented	Process-Oriented
	MRA	MD	MI	MT		Textual Language	Meta-model	Semantic Data-Model				
Decamerone	☒	☑	☑	☒	Assembly-based	MEL	MDM	MDM	☑	☑	☑	☒
MENTOR	☑	☑	☑	☒	Assembly-based, Paradigm-based	-	NATURE	-	☑	☑	☒	☑
MERET	☒	☑	☒	☒	Method Customization	Methodology Representation Model	ASDM	ASDM	☒	☒	☑	☒
MERU	☑	☑	☑	☒	Assembly-based	MRSL	MVM	-	☑	☑	☑	☒
MetaEdit+	☒	☑	☑	☒	Assembly-based	-	GOPPRR	-	☒	☑	☑	☒
MethodBase	☒	☑	☑	☒	Method Customization	Object Z	-	-	☒	☑	☑	☒
Method Editor	☒	☑	☑	☒	Assembly-based	MEL	UML	-	☑	☑	☑	☒

that each CAME environment has offered is evaluated and rated as its *Contributions*. We have also strived to provide a measure of the documentation available on each environment: If more than one author have published more than one paper on a CAME environment at different levels of detail, its *Available Literature* is marked as High; if more than one author have published papers on a CAME environment but they do are not much different as to their span and/or level of detail, it has been marked as Average; and if available publications on a CAME environment are rare or do not have the needed level of detail, it has been marked as Low.

In order to provide a more detailed analysis of CAME environments, we propose the ISO/IEC 9126 quality model [27] as a useful evaluation framework. ISO/IEC 9126 is one of a large group of internationally recognized standards applicable across a wide range of applications. We have instantiated the model to fit the CAME domain, and the CAME environments described above have been evaluated based on this adapted model, with the results tabulated for enhanced legibility.

Table 2. General analysis and comparison of existing CAME environments

Environment	Use	Number of Features	Contributions	Available Literature	Year of Introduction
Decamerone	Research	High	High	Average	1995
MENTOR	Research	Average	Average	Average	1996
MERET	Research	Low	Average	Low	1992
MERU	Research	High	High	Low	2001
MetaEdit+	Research and Commercial	High	Average	High	1994
MethodBase	Research	Low	Average	Low	1992
Method Editor	Research	Average	Average	Average	2003

3.1 The ISO/IEC 9126 Quality Model

ISO/IEC 9126 was originally developed in 1991 by the International Organization of Standards to provide a framework for the evaluation of software quality. However, ISO/IEC 9126 does not provide requirements for software, but defines a quality model which is applicable to any kind of software. This model defines six product characteristics which are further subdivided into a number of sub-characteristics (See Table 3). These characteristics and sub-characteristics constitute a detailed model for evaluating any software system. To be able to take different requirements of different systems into account, the model needs to be instantiated for each concrete domain by weighing the different characteristics and sub-characteristics accordingly.

3.2 A Quality Model for CAME Environments

Our quality model for CAME environments is an adaptation of ISO/IEC 9126; i.e. we have applied the model to the domain of method engineering. Table 4 illustrates our CAME quality model. The three characteristics of Functionality, Usability and Portability of the original quality model can be assessed based on the available literature; we have therefore focused on these characteristics. We use these quality characteristics and sub-characteristics to evaluate the CAME environments discussed earlier in this paper.

Table 3. ISO/IEC 9126 characteristics and sub-characteristics [27]

Characteristic	Sub-Characteristics	Definition
Functionality	Suitability	The presence of the required functions
	Accurateness	The correctness of the results
	Interoperability	Ability of software to interact with other systems
	Security	The ability of software to prevent unauthorized access
Reliability	Maturity	The frequency of failure by faults in the software
	Fault Tolerance	The capability of software to maintain its level of performance under stated conditions for a stated period of time
	Recoverability	The capability of software to resume working and recover the data after failure
Usability	Understandability	The effort needed for use the software
	Learnability	The easiness of learning how the software works
	Operability	The effort needed for operating the software
	Attractiveness	The quality of the user interface
Efficiency	Time Behaviour	The response and processing times
	Resource Utilisation	The resource utilisation
Maintainability	Analysability	The effort needed for diagnosis of faults
	Changeability	The effort needed for modification
	Stability	The risk of modification effects
	Testability	The effort needed for testing the modified software
Portability	Adaptability	The opportunity for moving the software to other environments
	Installability	The easiness of software installation
	Co-existence	The ability of software to coexist with other software systems in a common environment
	Replaceability	The effort needed for replacing other software
All characteristics	Compliance	The compliance of software with regulations and rules

Table 4. The CAME Quality Model

Characteristic	Sub-Characteristics	Criteria Description
Functionality	Suitability	▪ Evaluates if the CAME environment offers a suitable toolkit for the development of project-specific CASE tools. ▪ Evaluates if the CAME environment supports various SME approaches. ▪ Evaluates if the CAME environment supports process enactment. ▪ Evaluates if the CAME environment offers facilities to define semantics of method fragments.
	Accurateness	▪ Evaluates if ample knowledge is available as to the results of own tests or tests published by third parties that indicate the degree of effectiveness of the CAME environment.
	Functionality compliance	▪ Evaluates if the CAME environment supports standards and techniques such as: UML, XML …
Usability	Understandability	▪ Evaluates the level of understandability and usability of the interfaces. ▪ Evaluates the level of understandability and usability of the method representation language.
	Learnability	▪ Evaluates if the CAME environment has adequate documentation. ▪ Evaluates the level of learnability of the method representation language.
	Operability	▪ Evaluates if the CAME environment has graphical tools that facilitate the development of Method fragments.
	Attractiveness	▪ Evaluates if the CAME environment has attractive graphical design.
Portability	Installability	▪ Evaluates if the provider provides technical support and online help for the installation of the CAME environment.
	Co-existence	▪ Evaluates the capacity of the CAME environment to coexist with other independent CAME or MetaCASE environments in a common environment sharing common resources. For example, whether other MetaCASE tools can be installed to satisfy the CASE generation functionality.

3.3 Evaluation Results

The results are summarized into a matrix relating the characteristics and sub-characteristics to the features offered by the CAME environments reviewed (See Table 5) Deficiencies identified during the evaluation are indicated by a number, and an explanation is given in the legend below of how the system failed to meet the criteria in these cases.

Table 5. Evaluation of the CAME environments using the CAME Quality Model

	Quality Characteristics								
	Functionality			Usability				Portability	
Environments	Suitability	Accuracy	Functionality compliance	Simplicity	Learnability	Operability	Attractiveness	Installability	Co-existence
Decamerone	3,4	×	×	7	✓	✓	×	×	✓
MENTOR	4	×	×	✓	✓	✓	×	×	✓
MERET	1,2,3,5	×	×	✓	✓	-	×	×	-
MERU	3,4	✓	×	×	-	✓	×	×	✓
MetaEdit+	3,4,5	✓	✓	✓	✓	✓	✓	✓	✓
MethodBase	2,3,4	×	×	6	-	-	×	×	-
Method Editor	3,4	×	✓	✓	✓	✓	✓	×	-

Legend:
✓ Supported to a good extent
× Not supported
- Inadequate information to assess
1. Lack of CASE tool generation facilities
2. Partial coverage of the ME process
3. Inadequate support for SME approaches
4. Lack of semantic definition features for method fragments
5. Poor process support
6. Does not provide graphical meta-modeling language
7. Poor graphical meta-modeling language

4 Conclusion and Future Work

In this paper, we have summarized the main efforts performed in the development of CAME environments. Although CAME technology dates back to the early days of method engineering, it is not mature enough to support the whole process of situational method engineering. Each current CAME prototypes has its own advantages and shortcomings. In the following, the main shortcomings that current CAME technology suffers from are listed:

- Weak process enactment support: Even though product-related issues of ISD methods are fully considered and have been provided with computerized support, the process-related issues still need to be researched in order to find suitable ways for representing method process models. *Process Modeling Languages (PML)* [28, 29, 30] can be considered as suitable means for process representation; however, guidelines should be attached to a process described in a PML in order to support process enactment in actual ISD projects.

- Lack of support for situational method engineering approaches: The assembly-based approach is the only one adequately addressed. Paradigm-based and Extension-based approaches should also be supported by CAME environments.
- Partial coverage of method engineering process: Although method design and implementation phases are properly supported, there are still severe shortcomings as to support for method requirements analysis and method test.
- Method verification: Verifying the newly built method may be the last phase of the method engineering process, but it is never the least. Method verification requires a criterion set which a method can be checked against. But the difficult part of the task is determining how to perform the evaluation. Due to this difficulty, method verification is one of the hardest to automate. Current CAME prototypes perform method test through prompting feedback from the users of the method. Therefore, the newly developed method would not be verified until it is tested in an actual project situation.
- Weak method representation mechanisms: As mentioned in [31], there is no ultimate method representation language. Therefore, method representation languages are composed of fragments originating from several languages in a bid to obtain a purpose-fit language. This leads to a situation which is called *Method Engineering of Method Engineering Languages*. New method engineering languages need to be developed to support method verification.
- Lack of support for semantic definitions of method fragments: We believe that semantic meta-models should be an integral part of any CAME environments' Method Base, but few of the existing CAME environments address this issue. The lack of means for capturing and specifying the semantic aspects of method fragments leads to complications; examples are the selection and assembly of method fragments that may not be semantically composable into a method [32]. Describing the semantics of method fragments is one of the major problems in SME. To overcome this problem, method fragments need to be described in a complete and unambiguous way. However, as stated in [1], since methods and their semantics are interpreted differently by different human beings, there is no unique meaning for a method fragment. Nevertheless, method fragments can be anchored, i.e. described in terms of unambiguously defined concepts and relationships between those concepts, in a system for which the meaning is defined. Such systems are defined as ontologies in Decamerone and MERET.

Our future work focuses on the development of a CAME environment supporting the Hybrid Methodology Design approach [14]. This approach to methodology design uses alternative method engineering approaches for different parts of the process and at different levels of abstraction. It also provides an iterative and incremental framework allowing flexible application of four method development approaches, namely:

- *Instantiation approach:* with the focus on instantiating an already available process meta-model.
- *Artifact-oriented approach:* devising a seamless complementary chain of artifacts and building the process around it.
- *Composition approach:* using one of the already available libraries of process patterns.

- *Integration approach*: integrating features, ideas and techniques from existing methods.

Two of these approaches, *Instantiation* and *Composition*, are analogous to the *Paradigm-based* and *Assembly-based* approaches of method engineering, whereas the *Integration* and *Artifact-oriented* approaches are relatively novel in this context. The *Integration* approach is particularly nonconformist in comparison to usual method engineering practices, in that it promotes integrating ideas and techniques directly from existing methods, instead of first dissecting the methods into method fragments and then storing them in a method repository (as is common practice in the assembly-based method engineering approach); the motivation behind this stance is the observation that "breaking down the methods into fragments may result in loss of synergy and functional capacity" [14].

Acknowledgments. We wish to thank the Research Vice-Presidency of Sharif University of Technology and Iran Telecommunication Research Center (ITRC) for sponsoring this research. Also, special thanks to the anonymous reviewers of this paper for their helpful feedback.

References

1. Harmsen, A.F.: Situational Method Engineering. Moret Ernst & Young, Utrecht (1997)
2. Rolland, C.: A Primer for Method Engineering. In: Proceedings of the INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID 1997), Toulouse (1997)
3. Saeki, M.: CAME: The First Step to Automated Method Engineering. In: Workshop on Process Engineering for Object-Oriented and Component-Based Development, Anaheim, CA (2003)
4. Arni-Bloch, N.: Towards a CAME Tools for Situational Method Engineering. In: Proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications, Geneva (2001)
5. Dahanayake, A.N.W.: Computer-Aided Method Engineering: Designing CASE Repositories for the 21st Century. Idea Group Publishing, Delft (2001)
6. Kumar, K., Welke, R.J.: Methodology engineering: a proposal for situation-specific methodology construction. In: Cotterman, W.W., Senn, J.A. (eds.) Systems Analysis and Design: A Research Agenda, pp. 257–268. John Wiley & Sons, Chichester (1992)
7. MetaCase Consulting: Method Workbench User's Guide, MetaCase Consulting, Jyväskylä, Finland (2005), `http://www.metacase.com/support/40/manuals/mwb40sr2a4.pdf`
8. Saeki, M., Wenyin, K.: Specifying software specification and design methods. In: Wijers, G., Wasserman, T., Brinkkemper, S. (eds.) CAiSE 1994. LNCS, vol. 811, pp. 353–366. Springer, Heidelberg (1994)
9. Brinkkemper, S., Saeki, M., Harmsen, F.: A Method Engineering Language for the Description of Systems Development Methods. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 473–476. Springer, Heidelberg (2001)
10. Ralyté, J., Deneckère, R., Rolland, C.: Towards a Generic Model for Situational Method Engineering. In: Eder, J., Missikoff, M. (eds.) CAiSE 2003. LNCS, vol. 2681, pp. 95–110. Springer, Heidelberg (2003)

11. Gupta, D., Prakash, N.: Engineering Methods from Method Requirements Specifications. J. Requirements Engineering 6(3), 135–160 (2001)
12. Leppanen, M.: Conceptual Analysis of Current ME Artifacts in Terms of Coverage: A Contextual Approach. In: 1st Workshop on Situational Engineering Processes, Paris (2005)
13. Prakash, N., Goyal, S.B.: Towards a Life Cycle for Method Engineering. In: 12th Workshop on Exploring Modeling Methods in Systems Analysis and Design (2007)
14. Ramsin, R.: The Engineering of an Object-Oriented Software Development Methodology. Ph.D. Thesis, University of York (2006), `http://www.cs.york.ac.uk/ftpdir/reports/YCST-2006-12.pdf`
15. Tekinerdoğan, B.: Synthesis-Based Software Architecture Design. Ph.D. Thesis, University of Twente (2000)
16. Heym, M., Osterle, H.: A Semantic Data Model for Methodology Engineering. In: 5th Workshop on Computer-Aided Software Engineering, pp. 142–155. IEEE Press, Los Alamitos (1992)
17. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996)
18. Tolvanen, J.P.: Incremental Method Engineering with Modeling Tools. Ph.D. Thesis, University of Jyväskylä (1998)
19. Koskinen, M., Marttiin, P.: Process Support in MetaCASE: Implementing the Conceptual Basis for Enactment Process Models in MetaEdit+. In: Ebert, J., Lewerentz, C. (eds.) Software Engineering Environments, pp. 110–123. IEEE Computer Society Press, Los Alamitos (1997)
20. Koskinen, M.: Beyond Process Modelling Languages: A Metamodelling Approach to Customizable Concepts and Enactability in MetaCASE. In: Proceedings of the 4th Doctoral Consortium on Advanced Information Systems Engineering, Barcelona (1997)
21. Brinkkemper, S., Harmsen, F.: Design and Implementation of a Method Base Management System for a Situational CASE Environment. In: Proceedings of the 2nd Asia-Pacific Software Engineering Conference, pp. 430–438. IEEE Computer Society, Los Alamitos (1995)
22. Si-Said, S., Rolland, C., Grosz, G.: MENTOR: A Computer Aided Requirements Engineering Environment. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 22–43. Springer, Heidelberg (1996)
23. Plihon, V., Rolland, C.: Genericity in Method Construction. In: Proceedings of the 4th Asia-Pacific Software Engineering Conference, pp. 302–311. IEEE Computer Society, Washington, DC (1997)
24. Rolland, C., Plihon, V.: Using Generic Method Chunks to Generate Process Model Fragments. In: Proceedings of the 2nd International Conference on Requirements Engineering (ICRE 1996), pp. 173–181. IEEE Computer Society, Colorado (1996)
25. Plihon, V.: MENTOR: An Environment Supporting the Construction of Methods. In: Proceedings of the 3rd Asia-Pacific Software Engineering Conference, pp. 384–392. IEEE Computer Society, Washington, DC (1996)
26. Saeki, M.: Configuration Management in a Method Engineering Context. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 384–392. Springer, Heidelberg (2006)
27. International Organization for Standardization (ISO), International Electrotechnical Commission (IEC): ISO/IEC: 9126: Software engineering - Product quality; Parts 1-4. Geneva (2004)

28. Cugola, G., Ghezzi, C.: Software processes: a retrospective and a path to the future Software Process. J. Improvement and Practice 4(3), 101–123 (1998)
29. Zamli, K.Z., Lee, P.A.: Taxonomy of process modeling languages. In: ACS/IEEE International Conference on Computer Systems and Applications, pp. 435–437. IEEE Computer Society, Washington, DC (2001)
30. Zamli, K.Z.: Process Modeling Languages: A Literature Review. Malaysian Journal of Computer Science 14(2), 26–37 (2001)
31. Harmsen, A.F., Saeki, M.: Comparison of Four Method Engineering Languages. In: Proceedings of the IFIP TC8, WG8.1/8.2 working conference on method engineering: principles of method construction and tool support, pp. 209–231. Chapman & Hall, London (1996)
32. Brinkkemper, S., Saeki, M., Harmsen, F.: Meta-modeling based assembly techniques for situational method engineering. J. Information Systems 24(3), 209–228 (1999)