

# Formal Metamodelling and Agile Method Engineering in MetaCASE and CAME Tool Environments

Eleni Berki

Department of Computer Science and Information Systems,  
University of Jyväskylä, Mattilanniemi Campus, AGORA Building, P.O. Box 35,  
Jyväskylä, FIN – 400 14, Finland  
Tel.: + 358 (0) 14 260 3036, Fax: + 358 (0) 14 260 3011, E-mail: eleberk@cc.jyu.fi

**Abstract:** This paper explores the capabilities of formal metamodelling in order to support agile process models in the context of CASE-Shells and CAME tool environments. The strengths and weaknesses of metamodelling environments are discussed, while their limitations to metamodelling at the level of both software process (method engineering) and product (methods metaspecifications) are critically examined. The paper provides examples by focusing on MetaEdit+, a MetaCASE multilevel tool of the CASE-Shells paradigm. A coherent proposal is presented on what is needed to suitably adapt and customise MetaCASE and CAME to new organisational needs and extend their functionality to model processes agilely and formally. In particular, it is demonstrated how IS development methods that are specified in CAME environments can be extended in a manner that they are more agile, dynamic, richer in syntax and semantics and cater for computation, testing and successful process evolution.

**Keywords:** *Process Metamodelling, Method Engineering, MetaCASE (CASE-Shells), CAME (Computer Assisted Method Engineering), Finite State Machine (FSM), Computability, Testing.*

## 1. Introduction

The needs for customised, flexible and agile methods that satisfy the requirements for stakeholder involvement and user participation has led software designers to invent metamodellers [1, 2, 3]. This underpinned the need for the method user and method modeller to reflect on their own intuitions and construct their own desirable methods. As a discipline, Method Engineering (ME) considered the solution to bridge the various gaps that traditional non-agile design methods created. The subsequent creation of CASE and CASE-Shells (MetaCASE) tools was also directed by these needs [4, 5]. Nowadays there are groups of people (method, application, software and process engineers) who use CASE-Shells to create their own knowledge-oriented methods, to document their own tool environment design their software development processes [6]. The current popularity of CASE, MetaCASE and CAME environments has spurred numerous activities to broaden their use and support them within the software research and development community in academic institutions, in industry and in organisations [7].

However, the metamodelling facilities offered by MetaCASE tools did not provide adequate metadesign patterns in order to be able to alleviate methodological deficiencies. For instance, *none* of the existing CASE-Shells offers direct method implementation and *none* facilitates testing. Implementations corresponding to specifications developed by methods that facilitate testing have the advantage of being amenable to further analysis and to formal testing for quality assurance [8]. Regarding the software development lifecycle stages, very few methods and subsequently their automated CASE and MetaCASE tools deal with implementation. Most stop at physical design stage. The developer is given no guidance on implementation strategy, choice of platform or programming language.

The requirement for testing of metamodelling processes and products and the demand for tool integration (industrial but not commercial priority) add to the challenges that MetaCASE technology faces nowadays. Testing the development of software systems [9] has always been one of the most challenging and most difficult areas in software engineering [10]. *"Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and coding."* [11].

Several of the leading (mostly O-O) software design notations and methods have added support for the modelling and representation of the metadesign architectures of CASE-Shells tools. Thus, many software development tools and environments have inherited similar support because of the concepts, structure and syntax reuse [12]. The basic metamodelling architectures that are used in MetaCASE are based on 'static' data models (mostly Entity Relationship Diagrams) and on their extensions. Hence the basic limitations that appear in the accurate expression of a method are semantic and syntactic weaknesses, which are inherent in the generic structure of the data model itself. Unavoidably, the most important semantic weakness is the inadequacy of the tool's metamodelling power to express "the modelling data" of dynamic and interactive elements [13].

Simple modelling approaches that do not obey to the static rules of a metamodel such as CoCoA (ComplexCoveringAggregation) and NIAM (Nijssen's Information Analysis Method) address dynamic issues of methodological specifications semantically and syntactically better. Many different data models such as Object-Z have been utilised to address specific method engineering limitations and other have also been used for method integration such as NIAM, but unfortunately they are not implemented to be used as MetaCASE tools [14].

Other problems of metamodelling with MetaCASE tools that are associated to the static structure of "*metametamodelling*" architectures are the lack to express the process of metamodelling a method and the "explosion" or "refinement" of a method's constructs. These are sometimes semantically handled with pseudo-dynamic extensions in *metametamodelling* architectures, which, in turn, is a barrier for the correct transformation of method knowledge; and therefore limits the potential applicability and modelling power of the derived method model itself. Additionally, the semantics and syntax of a method cannot not always correspond

to the modelling of the pragmatics. The lack of expressing the correct meaning by metamodelling points additionally to the difficulty of integrating and standardising among different methods and tools. By demonstrating that representations of different application domains can have one interpretation in terms of one design and one implementation could provide ways of better comprehension of the conceptual process modelling and improved ways of using the available information of that correspondence [15].

Many metamodellers and practitioners realised the need to review the established ME concepts [16]. The interconnected nature of these problems has very little been emphasised [8, 13] and hardly any suggestions pointed to *agile* and *generic* metamethods' specifications, which could provide ME and CAME with both generalised and specialised approaches, designed to offer consistency at many metamodelling levels. An agile and formal metamodel would lead to an improved understanding of methods through problem solving and method construction.

## **2. The Need for *Agile Method Engineering (AME)* in CASE-Shells**

A detailed classification with descriptions of many of the previously mentioned problems are encountered in [5, 14, 17]. A rich list with full description and more detailed analysis can be found in [12, 18, 19] where the first attempts to provide MetaCASE processes with adequate models at *metametamodelling* level are also mentioned. The following is a summary [13] regarding the modelling inefficiencies in CASE-Shells and other observations that point to the need for an agile pattern for flexible method construction:

*Integration and Expressability of Data and Process Dynamics of Method Modelling:* Methods (however not all their techniques) are process models, which handle data and transform it continuously; this being their nature, they are changeable and their dynamic structure affects both data and control processing [13, 14]. The static and dynamic conceptual constructs of the methods should be sufficiently generic, expressive and agile in order to specify and monitor changes. Thus, their instances should allow a number of distinct design architectures, which continuously evolve and expand, to be transformed into more dynamic [19] and integrated ones [20]. The MetaCASE architectural patterns should clearly focus on defining the interaction of the method's components.

*Computability and Implementation Issues in Method Metamodelling:* Generally speaking, systems implementations represent initial specifications but without corresponding to any computational characteristics of the design [21, 22]. Since development methods do not have guidelines for further programming, a metamodel should be able to provide them with encapsulated rigorous syntax and semantics. This will allow a physical design to be mapped to a target implementation language with less effort and productivity loss. The projection of features such as the method's computational properties are highly desirable because they aid communication for systems documentation and thus facilitate the collaboration among developers and maintainers teams.

*Testing of Methods and of the Process of Modelling a Method in MetaCASE: A metamodel of a MetaCASE and CAME tool should provide methods and their instances with well-defined rigor, in such a way that software errors (system and software design errors) to be avoided or to be detected earlier in the life cycle. Syntax, semantic, logic and algorithmic errors could be eliminated if the process model in the metaspecification environment was populated with techniques that provide testability for cross-checking of the artefacts' correctness [13, 23].*

### **3. The MetaPHOR Initiative and the MetaEdit+ Tool**

The main goals of the MetaPHOR (Metamodeling, Principles, Hypertext, Objects and Repositories) research group have been to develop principles and models to serve as practical and flexible solutions for IS and ME domains, and to investigate, design, implement and evaluate ISD tools and methods. The primary problems addressed in the research are the development of new ways of modelling adequate ISD methods, their evaluation in different circumstances, and the design and implementation of support tools for them. In tool building the underlying principle is to develop method-independent solutions and architectures and adapting methods for different contingencies. The issues of method integration and agility and ways to achieve them had always been of great importance.

MetaEdit+ is a MetaCASE and CAME tool, which has been used widely in order to model systems and metamodel methods. It is a multi-tool, designed to assist many users with diverse knowledge backgrounds and different skills. It incorporates modelling and metamodeling facilities and has recently also been enhanced with the ability to *metametamodel* method specifications [12]. MetaEdit+ is a CASE-Shell (MetaCASE) tool that was evolved from MetaEdit, a simple CASE tool, which provided automated support for systems analysis and design. MetaEdit+ is a multi-method and a multi-tool platform for both CASE and Computer Aided Method Engineering (CAME). *“As a CASE tool it establishes a versatile and powerful multi-tool environment which enables flexible creation, maintenance, manipulation, retrieval and representation of design information among multiple developers. As a CAME environment it offers an easy-to-use yet powerful environment for method specification, integration, management and re-use...”* [24].

*The Evolution from MetaEdit to MetaEdit+:* The expanding needs of MetaCASE technology pointed out that the methodological constructs expressed by OPRR, the main metamodeling architecture [25], were not sufficient to express the next generation of MetaCASE and CAME tools. New methodological issues and concepts needed to be captured and modelled adequately with the use, and sometimes reuse of the older components. In the case of MetaEdit+, the expansion from OPRR to GOPRR followed a smooth transition obeying to reuse and extension rules of the OPRR constructs, which are explained in the next section. The new version, GOPRR, was extended with the construct of the *Graph* (or *Table* or *Matrix*), provided metamodelers with the necessary abstractions and the adequate visualisation to express a method as a Graph, Table or Matrix. This richer

representation of methodological constructs addressed and expressed the modelling needs for future MetaEdit+ applications.

### 3.1 MetaEdit+: The Architecture of the Method Workbench

The Method Workbench is a significant part of MetaEdit+ tool. GOPRR is the basic architectural structure that is used to create the products of all levels, i.e. methods and their instances. GOPRR (like OPRR) recognises in a method's generic structure (and therefore in its instances) the semantic concepts of *Objects* and *Relationships*, which both possess *Properties* and *Roles*; all these constructs can be viewed in GOPRR as a *Graph, Table or Matrix*.

When creating new method specifications in MetaEdit+, the metamodeller should firstly concentrate on the “constructs” of a method. In doing so, she/he must use the GOPRR metamodel to guide the whole metamodelling process. According to the philosophical rules of GOPRR, any method could be represented as having the following (concrete and abstract) constructs, whose initials form the acronym GOPRR:

*Graphs*, which are sets of objects and their connections;  
*Objects*, which are identifiable design entities in every technique/method;  
*Properties* are attributes of graphs, objects, relationships and roles;  
*Relationships* are associations between objects;  
*Roles* define the ways in which objects participate in specific relationships;

**The Method Modelling Process Using GOPRR:** Any method can be defined according to the *definitional constructs* of the GOPRR architecture, which are Graphs, Objects, Properties, Roles; all these are parts of Relationships between objects. These form the ‘conceptual structure of a method’ and it is only when these have been established that it is possible to proceed with formulation of the method notation and subsequent method construction.

The following process of method definition and construction is described in MetaEdit+ Version 2.5 Method Workbench User's Guide:

- A. Identify and define the object types of the method.
- B. Define the properties of the object types.
- C. Identify the relationship types of the method.
- D. Define the properties of the relationship types.
- E. Define the role types of the method
- F. Define the properties of the role types.
- G. Define the necessary symbols for objects, relationships and roles.
- H. Define the graph types and add the object, relationship and role types into them.
- I. Define the bindings of the relationships in the graph types.
- J. Define explosions and decompositions of the object types in the graph type.

These steps can be performed in an iterative way and partly in parallel, and the definitions of the types can be modified later, but this is the rough model of the method modelling process.

### 3.2 Syntactic Strengths and Semantic Limitations of this ME Approach

GOPRR is an O-O metadesign architecture; its implementation in MetaEdit+ has been greatly improved because it obeys the rules of inheritance, polymorphism and reuse. The dynamic properties of objects allow their successful implementation with Smalltalk, which is a programming language that fits the needs for implementing reactive systems with dynamic requirements. The GOPRR metamodelling architecture has sufficiently generic constructs, which makes it suitable for the design of methods. This is also the main reason that it serves as a tool architecture for other CASE-Shells environments such as RAMATIC in Sweden [26].

The basic disadvantage of this approach is that resulted methods cannot be tested formally because no formal testing procedure is incorporated in the metadesign process. Moreover methods are not viewed as process models because the specification of GOPRR itself lacks the ability to adequately express the dynamic constructs of a method. The fact remains that GOPRR is based on an extended Entity Relationship Diagram and as such, it inherits its static structure. Following the semantic definition of a method using GOPRR, a method is defined as the “*Cartesian product of the sets of objects, roles and properties that a method consists of, together with the initially specified connections and relationships between them*” [13]. The specification and implementation of a method in MetaEdit+ is facilitated by the *Binding* construct, which extends and enriches the flexible (but fuzzy) conceptual definition of the Relationship construct. Therefore, there is no ‘appropriate’ semantic construct in GOPRR that would allow the expression of *dynamic* methodological aspects. For instance, changes in the resulting method specifications could not be modelled during their application [18].

Reflecting on the previous analysis, we may support the following: In order to improve the quality of method specifications and their subsequent tool implementations, a more efficient and more generic approach for modelling methods is needed. It is also important to establish in a scientific and cognitive way whether the adoption of a specific method will be able to model the static, dynamic and computational requirements for a system, and to what degree. This is a question that can be answered only if we could decide on a method’s constructs during and/or after its use by viewing them at a different level of diagrammatic and theoretical abstraction. That is, by the use of an effective and suitable metamodel that would allow us to visualise them, reason about them and finally reuse them after proper formal testing procedures and re-engineering principles.

It is obvious that while metamodelling and ME principles were associated with the needs of CASE technology, MetaCASE gave rise to more demanding modelling environments. It underlined the need for more abstraction skills to navigate through

different metamodelling levels whilst preserving the notational and semantic consistency of the associated artefacts of every level [7]. The expansion and use of CASE-Shells technology also assisted in realising the need for more co-operation and interaction among teams, indicating clearly the needs for tool and method standardisation and integration.

Emphasis has also been given to the inadequacy of capturing computational characteristics of metamodels and the dynamic knowledge of a method, which is found in methodological data and processes' transformations, heuristic rules and recommendations, methodological constructs' explosions and refinements, dynamic rules of naming and so on. For instance, in case of MetaEdit+ tool, it is explicitly stated that "*none of the metamodeling techniques support grammar specification for formal textual descriptions, such as process specifications, data dictionaries or textual grammars. These are especially important to better integrate modeling tools and models into other tasks of ISD, such as generating prototypes, program code, or visualizing available data and program structures.*" [14].

Thus, we will examine and apply formal concepts to alleviate limitations of metamodelling techniques in MetaCASE environments. Furthermore, we show how such methodological issues of computation, testing, integration [8] and properties of communication and representation [2, 27] can be achieved at the level of *metametamodelling*.

## **4 A Formal and Agile Way of Metamodelling Methods**

We considered the family of formal dynamic models of Finite State Machines (FSMs) and *general machines* [28] as *process metamodels* that aid to alleviate the previously identified limitations through diagrammatic and theoretical abstraction [8, 13]. We remodelled OPRR and GOPRR as machines models. We demonstrate our outcomes using examples of this new type of metamodelling, which results in providing testable, computational and evolutionary method models.

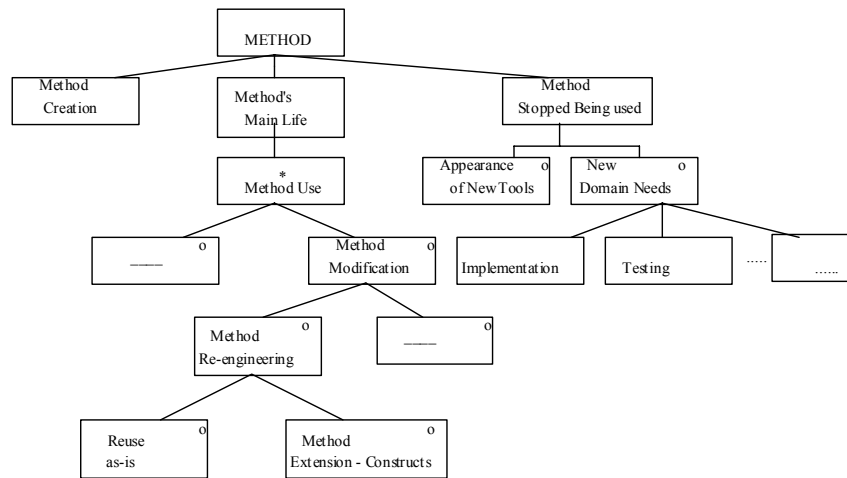
### **4.1 Expressing and Understanding Methods in Terms of their Dynamics**

The lifecycle of a method represents how a method is created, used, reused, maintained with re-engineering [14] and finally is archived, if it is no longer required for modelling. Our conceptualisation of a method as a dynamic entity is given next (Fig. 1), using JSD notation to depict its lifecycle. This indicates clearly the events that affect a method's life changes, and how they occur, that is in *sequence, iteration, selection or multiple selection* [8].

This diagrammatic conceptualisation of a method shows that the data of the method undergoes many changes depending on the various events that happen in its environment and affect it. We selected some of these events and chose to model them in JSD notation, which is a suitable notation to show dynamic interactions. These events may be *internally triggered*, which basically have to do with the modelling process by the method's modellers such as 'Method Modification', or

externally triggered such as 'Appearance of New Tools' in the market or time triggered such as 'Method Licence Expires', and so on.

All these events are interdependent (Fig. 1), and consequently affect the method's status; for example the 'Appearance of New Tools' may trigger the 'Method Modification' event. However, not all these interactions are shown clearly with JSD notation; even the modelling of these events has to be facilitated with mid-pseudoevents, as, for instance, in the case of 'Method Re-engineering', which needs to be facilitated in its modelling by 'Method Modification' and this in sequence by 'Method Use'.



**Fig. 1: Method Modelling** - The entity life-cycle of any development method

This representation of a method clarifies the notion of a method as a dynamic object but it also inherits the modelling problems of the JSD notation. The next step is to describe how we can map the *generic semantic and syntactic constructs* of a method to a formal machine model. Subsequently, we show how to construct its isomorphic diagrammatic representation.

The family of 'machines' is very rich both in diagrammatic structures and evolution semantics and very expressive in potential pragmatics modelling. Hence, various method models should be able to be expressed as alternative isomorphic models in this class of computational rigour.

## 4.2 An Agile Theoretical Construction as a Generic Formal Model of a Method

Since a method is a process model, its dynamic nature can be described and depicted in such a notation that its syntax and semantics be directly mapped to FSM and general machine notation [13].

If a  $\lambda$ -NFA  $K = (Q, \Sigma, \delta, s, F)$

then a method  $M = (Q', \Sigma', \delta', s', F')$  can be defined as a computational structure with the following semantic constructs:

$Q' = \{\text{an alphabet of stages which contain information on the technique/method being modelled; this sometimes can indicate information on an incomplete method diagram under construction}\}$

$\Sigma' = \{\text{an alphabet of formation instances}\}$ , a *formation instance* is the construct with the participation elements from the next possible alphabets, subsets of  $\Sigma'$  which represent the requirements

$\Sigma'_1 = \{\text{an alphabet of events}\}$   
 $\Sigma'_2 = \{\text{an alphabet of data}\}$   
 $\Sigma'_3 = \{\text{an alphabet of properties}\}$

...

and so on, in case we want to *extend* the method's *morphological features* and utilise *new* methodological generic constructs that are different in nature and semantics from the above and therefore belong to different categories, that is different alphabets.

Additionally, the method's dynamic definitional construct can be smoothly mapped to the transition relation/function of any type of machine model. So,

$$\delta' \subseteq Q' \times (\Sigma' \cup \{\lambda\}) \times Q'$$

is the transition relationship, which takes as input a combination set of all (or some) of the members of the above alphabets (or different, depending on the technique) and transforms it to the next stage. The result of the state of the particular transformation will be recorded in the state of the graph before the next transformation takes place, with the output of the previous one(s) to behave as the input in the next transitional function.

The following two mappings are also important for method modelling because they define the start and the final states of a method. Thus, issues of computability, completeness and testability are always of importance in CAME environments as well as for the instances of the methods and their implementation.

$s' \in Q'$  is a start state i.e. the technique in its initial state, before any design construction takes place.

$F' \subseteq Q'$  is a set of theoretically final states that a method can reach.

Theoretically, there are also *unreachable* states, and this means that a method can never reach these states. Careful thinking of this aspect could associate it to the *testing of method construction process*, which is an issue to be addressed at metametamodelling level.

### 4.3 A Graphical Representation of a Generic Method Architecture

The pictorial representation of the conceptual structure of methods facilitates the understanding of their constructs. So, in order to sufficiently represent diagrammatically the previous methodological constructs of the family of machines, we hereby present the diagrammatic constructs that can participate in methods and in their instance formation. More details on the theoretical foundations of these syntactic constructs and their applicability in various case studies can be found in [8, 13, 29, 30].

The main focus at this stage of the representation is to attempt to clarify, relate and finally communicate adequately the system knowledge that is captured in a method. In doing so, we will map the concepts of the MetaEdit+ modelling architecture, i.e. GOPRR to those of an FSM. The mapping is also necessary to indicate the agile ways by which this representation can be used in order to semantically capture the domain knowledge.

### 4.4 Modelling Agilely a Method and the Process of its Modelling

Following our new theoretical and diagrammatic knowledge of what constitutes a method (Fig. 1), we next present in Fig. 2 the equivalent isomorphic FSM model (metamodel) of the generic model of *any* method as it was depicted using JSD notation in Fig. 1. After our alternative way of metamodelling with FSMs, a method and its modelling process can *both* be expressed formally and dynamically, reusing the same notation (for method and process modelling integration).

#### *The Conceptualisation of a Method as a FSM*

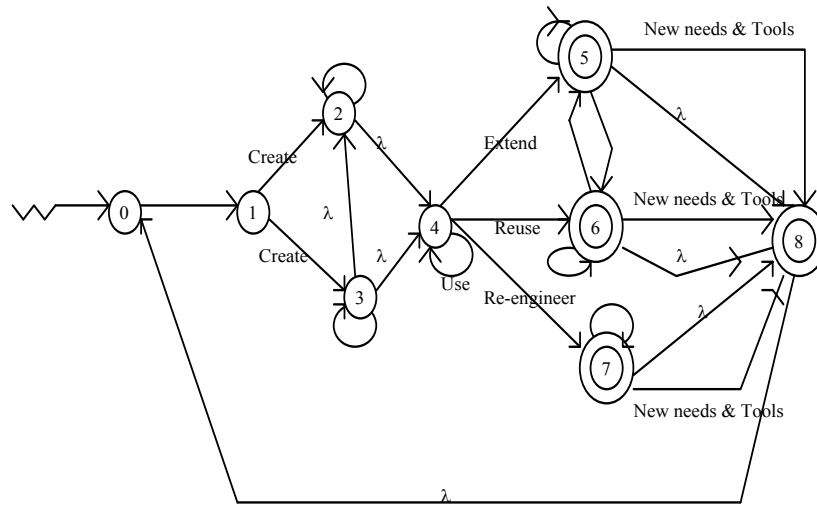
Firstly, Fig. 2 shows a *formal diagrammatic representation of a method* in terms of a Finite State Machine, which corresponds to the theoretical method representation that we described earlier, in which a method is described as a dynamic entity.

This is a more compact model, which does not need to model pseudo-events to facilitate method modelling such as in JSD notation in Fig. 1. It depicts only the 'real' events and shows clearly their interactions and how they affect a method. Furthermore, these can be expressed as *regular expressions* and therefore a

method's generic specification(s) and instances can have all the benefits of formality, computability and testability [13].

Following the consistency of the notation for the isomorphic model of a method as a Finite State Machine, we can state the following:

Since the FSM transitions model the events that affect a method M throughout its life, then consequently the FSM states, which belong to  $Q'$ , model the status of the method M after being affected by a particular event. For instance, in Fig. 3 some of the isomorphic diagram's information can be presented as follows:



**Fig. 2: Method Modelling**–The formalised generic model of a method, as model for the ‘target process’

The events that affect the method M belong to set  $\Sigma'$  (= an alphabet of M's events) and this set can be denoted as

$$\Sigma' = \{\text{Create method, extend method, re-engineer method, ..., } \lambda\}$$

The states of the life of a method M (labelled only as natural numbers in the diagram) belong to set  $Q'$  (= an alphabet of M's states); this set can be denoted as

$$Q' = \{\text{State 0, state 1, state 2 = generic method created, state 3 = instantiated method created, ... state 7 = re-engineered method, state 8 = archived method}\}, \text{ with}$$

$s' = \text{State } 0 \in Q'$  as the initial state of M,

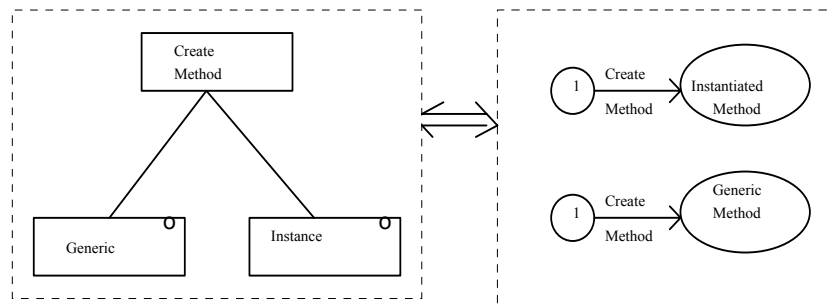
meaning that nothing has happened yet to alter the state of the method (no graph, table or matrix exist) and

$F' = \{\text{State } 8 = \textit{archived method}\} \subset Q'$  as the set of final states of M,

which means that some event(s) affected M not to be in use any more.

Such events might be *new needs and tools appear* (externally triggered), *method licence expires* (time triggered), *metamodeller changes notation* (internally triggered) and so on.

The events that cause the state changes in a method's life are plenty. A method can be affected by many events during its lifetime so we just chose a few characteristic ones to name and depict in Fig 2. In fact we put some unlabelled transitions, which may also model some events that may happen and affect a method's life. The same is true for its states: some event, which is not included here may happen in the future and that may cause a method to change its state to an unknown at present state. Such a case can be seen in Fig. 3.



**Fig. 3:** JSD and FSM equivalent notation: Creating a method may result either at the graph of its *generic* or the graph of its *instantiated* structure.

Therefore, the sets, which represent the alphabets of states and events may have many more members than the events and states that we chose to represent here. Fig. 3 presents in JSD and in the equivalent FSM notation the event *create method* (for method creation). In practice, this event may be either *create generic method* or *create instantiated method* (especially in MetaCASE tool environment). The resulted state for the method's status will be either the instantiated graph of the method or the generic (polymorphic) graph of the metamethod.

The previous JSD and FSM methodological constructs can just be added to the diagrams presented in Fig. 1 and 2 respectively. The reuse, extensibility and consistent expansion between different levels of detail (as will be seen in the next figures) of this dynamic notation is a major advantage when utilising these metamodelling concepts in CASE-Shells environments. In this type of FSM diagram we do not need to use state indicators for selection, sequence or iteration of events since these are encapsulated in the syntactic details of the diagram. Moreover, it includes a complete method/technique modelling of an instance. In modelling an instance of a method, our information on events and initial, final and other states is more definite and the modelling could be complete.

Utilising the same construction rules that are mentioned in this section, and taking our metadata from GOPRR metamodel and the process guidelines to construct a method (section 3.1), we also *metarepresent* the *process* (or *metaprocess*) of *modelling a metamethod* (metametamodel) with GOPRR in the Method Workbench of MetaEdit+. This is depicted in Fig. 4.

#### **The Conceptualisation of ‘the process of modelling a method’ as a FSM**

Reusing the same notation and theoretical model of the FSM method representation, in which a method itself is addressed as a process model, we utilise it once more in a more abstract level (next metalevel) in order to depict ‘the way of modelling a method’. Fig. 4 shows a *formal diagrammatic representation of the process of modelling a method* in terms of a FSM.

In Fig. 4, we represent the modelling process that takes place when we model a method with the Method Workbench of the MetaEdit+. The arrows with labels A, B, C, ..., J correspond to the functions A, B, C, ..., J of section 3.1, that is:

- A = *Identify and define the object types of the method;*
- B = *Define the properties of the object types;*
- ... ..
- J = *Define explosions and decompositions of the object types in the graph type.*

The main concept and philosophy of this representation of the process of modelling a method is the following:

The arrows (transitions of FSM) depict the activities (or transactions or events) that take place for the modelling. Again they comprise an alphabet  $\Sigma'$ , where the states of FSM correspond to information given on the particular stage of modelling. The State alphabet is  $Q'$ . The set of final state(s)  $F' \subset Q'$  indicates that the method’s modelling process has been completed. The last state, for instance, can be called ‘*method completed*’, which means that the generic structure (or an instance of the method modelling a particular situation) has been ended.

More analytically, we can attribute the above to the following mathematical notation, according to the method's formal description (FSM isomorphic model) as specified in section 4.2:

$\Sigma' = \{A = \text{Identify and define the object types of the method}, B = \text{Define the properties of the object types}, C, \dots I, J = \text{Define explosions and decompositions of the object types in the graph type}, \lambda\}$ .

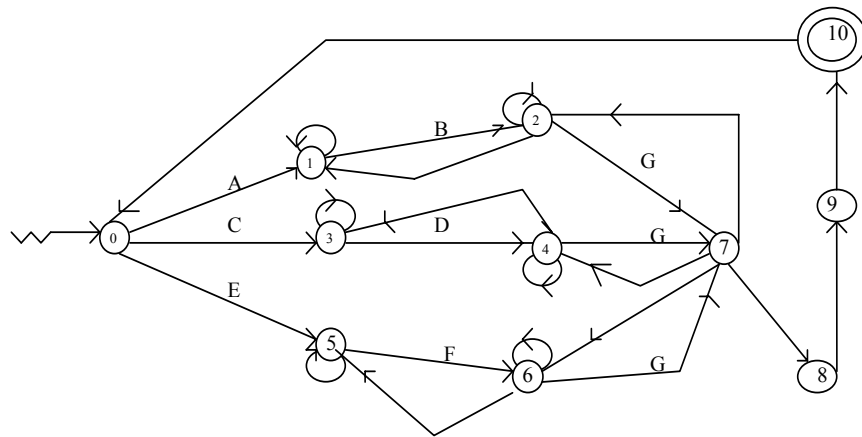
$Q' = \{0, 1, 2, \dots 10\}$ , where indicated state descriptions could be the following, according to the effects of the particular events:

$s' = \text{initial state} = \text{State } 0 = \text{method at its initial status} - \text{no event has taken place yet to form any construction}$

State 1 = 'method status with object types defined' in the graph/table/matrix

...

$F' = \{\text{State } 10 = \text{'method completed'}\} \subset Q'$  after defining all explosions and decompositions of the object types in the graph/matrix/table



**Fig. 4: Method Metamodelling** - The formalised generic model of a 'process of modelling a method'

Hence, reusing the same notation and constructs we re-modelled again the "process of modeling" and depicting a method; thus, we achieved 'notational' integration for both ways of modelling.

#### 4.4 The Meta-Agileness (!) and its Meaning in terms of Method Engineering

Tardieu states that dynamic modelling should explore the capability “*to express both the target process and the development process*”. He continues supporting that: “*This new challenge is both exciting and important, because it may lead us to information systems which could evolve in their structure as they are operating*” [31]. The last three diagrammatic models of Fig. 2, Fig. 3 and Fig. 4 realise these requirements, and obey to a formal conception of a method as a process model, according to the new theoretical knowledge of what constitutes a method.

Under this type of process modelling, the generic structure of a method can be any *morphologically complete* construct of the method, otherwise called a ‘particular’ technique of the method. That might be, for instance, a graph in case of ERDs, DFDs, Object or Class Diagrams, a table in case of Decision Tables, a Schema’s textual description in case of Z-method and so on [13]. As can be seen, the abstraction rules are specific and quite general for metamodelling agilely a wide range of specifications [8].

In different metamodelling environments (automated or non-automated) the activities of modelling may be different according to the emphasis on the modelling constructs. For instance, if the method’ s or tool’ s metamodelling architecture is other than GOPRR, then the modelling process with a Finite State Machine can be customised and adjusted to the guidelines given to follow that particular metamodelling process. In addition, both method and its modelling process can be further computed and tested because of the syntactic and semantic strengths of FSM notation.

The advantages of this notation is its flexibility and adaptability to accommodate changes. This means that if we want to add any process modelling guidelines or any new modelling states, then we just reuse the same specification structure and just add the new constructs to the FSM isomorphic graph, by simply making the suitable connections to the already existed states and transitions. This facility is extremely useful when CASE and MetaCASE tools undergo changes and need to be extended and integrated [34, 35] in order to facilitate the accommodation of new modelling paradigms; and therefore extend their own functionality, testability [36] and *metaagility* (!) in order to become more competitive and more marketable.

### 5. Conclusions, Ongoing Research and Future Directions

We referred to the capabilities of the MetaCASE tools technology and examined MetaEdit+, a MetaCASE and CAME tool. We further examined the semantic and syntactic constructs of GOPRR, the main architectural metamodel for modelling methods and their instances in MetaEdit+ and in other CASE-Shells modelling environments. The following have been identified: (i) lack of expression of both data and control, (ii) the need to express the computational characteristics of methods for facilitating future implementations and (iii) the redesign of the generic structure of methods in order for them and their instances to be agile and testable.







