

Database engineering processes with DB-MAIN

D. Roland, J-L. Hainaut, J-M. Hick, J. Henrard, V. Englebert

FUNDP, University of Namur
rue Grandgagnage 21, 5000 Namur, Belgium
dro,jlh,jmh,jhe,ven@info.fundp.ac.be

Abstract - Software engineering needs more and more to be supported by CASE tools. Since databases are at the heart of information systems, they deserve a particular attention. More and more CASE tools allow method engineers to implement their own methodology and they allow users to record all their actions, with their rationales, in order to improve the quality of the design and the quality of the documentation of the design. DB-MAIN is such a database oriented tool with a method description and a documentation generation facilities. But it has its particularities like its procedural non-deterministic Method Description Language, its well integrated multilevel histories and its user-friendly methodological engine.

I. INTRODUCTION

The need for controlling software processes has long been recognised as a strong requirement for improving the quality of software development and maintenance, just to mention two of the most critical problems. Database engineering is an important domain of information system engineering that deserves a particular attention since a database is at the heart of every information system. Controlling software processes consists in analysing the engineering activities that have been carried out to infer useful information on the products, the processes and the actors. Clearly, this control must be based on some trace of the engineering activity. The trace, or (better) the history of a software process consists of the recording of all the activities, all the decisions taken and all the rationales that occur during this process. To be useful a CASE tool has to guide a database engineer all along the process while recording the history during all phases; and it has to do it the most user-friendly and helpfully possible.

A. Process modelling

Database engineering processes can be modelled at a very fine grain. Though database design methods have been fairly standardised, developers like to follow their own way of working when they are faced with non-standard problems. This is the case for reverse engineering, but every database engineering activity can, sooner or later, require a high degree of flexibility in the way problems are solved. Though, developers still require methodological guidance, but according to their own methodology. Hence the need for powerful but flexible process models and for tools that are able to enact them. That is why we proposed in [24] an executable modelling language (MDL) which is both procedural and non deterministic for describing database engineering methods. When using an MDL method, the supporting CASE tool will explain the analyst what to do or how to do it. The CASE tool will possibly perform well defined processes that can be auto-

mated, or present to the analyst the list of processes that can or have to be performed and let him do the job in a semi-controlled way.

B. Building a documentation

Every action performed during a database engineering project can be recorded in the *history* of the project. Further maintenance, migration or extension of the software product will largely profit from this history. For example, the rationale of a decision cannot be ignored when trying, later on, to modify the components resulting from that decision. This history will be both *readable* by humans for documentation and *formal*, to be analysed and reused by the CASE tool. Indeed,

1. a history generally will need some polishing before being usable (trimming it from traces of trial-and-errors, from discarded branches, from loops, etc.)
2. useful information can be extracted from a history (design quality, auditing, skill, design heuristics, resource allocation, timing, etc.)
3. new derived histories can be obtained, such as a fictive forward history built by inverting the history of a reverse engineering process [12]
4. propagating design changes through the design products can be automated, or at least assisted, by replaying the history of the former design.

C. Supporting CASE tool

Process modelling would be an academic activity only if no CASE support were not provided to control the processes. The approach proposed in this paper is being implemented in the DB-MAIN CASE tool. This work is carried out in the DB-Process framework, a joint project of DB-MAIN R&D programme ([9],[13]).

D. Position of the paper

Along the years, several research teams did and still are doing work on process modelling aspects. Among these, we can name the DAIDA project [16], NATURE [20], MENTOR [7], PROART [21], then PRIME [22], the team of J. Souquères [27] who mainly focus on requirements engineering. We can also cite [2] which is more oriented toward software development process modelling. Business process modelling is supported by, among others, Adele [6], Process Weaver which allows the designed processes to be followed and is/Modeler, ProcessWise, IDEF [18], SilverRun or SOCCA [3] which are just descriptive tools.

A common point between most of them is the use of a declarative language or a graphical language and the need for an

inference engine to use their methods. All these views are at the opposite of ours since we use a semi-procedural scheme. Some of these projects record the activities that are performed as well as the rationales. In [23], C. Potts and G. Bruns explain why and how to record the reasons for design decisions. Reference [6] explains how processes can be supported, with the recording of what is actually done and with versioning, but stays at a high level; we know what tool has been used, with what products, when, but we do not know what happened during the use of the tool. [22] goes beyond by trying to better integrate all the tools used.

On another side, meta-CASE like MetaEdit+ [17] or Kogger [4] allow industries to personalise their tools, but it remains mostly on the graphical level; even when processes are taken into account, it is rather simple.

In this paper, we describe how DB-MAIN allows a user-defined methodology (defined with the MDL language) to be followed and how all the performed actions are logged.

E. Contribution of the paper

We will see, with the DB-MAIN CASE tool, how a generic CASE tool can be methodologically personalised. We can allow a method engineer to develop and implement his/her own method. The tool, according to the method, can assist a database engineer in his/her job. At the same time, all the actions of the engineer can be logged. Since we use a very well integrated tool, the log of every processes are well integrated; we can know exactly what happened at every moment. This allows us to reuse the history not just for replay but also for a broad range of purposes.

We will also see how the particularity of a procedural language to describe methods can be very beneficial to the engineer in the way the method is presented to him/her.

II. PROCESS MODELLING WITH THE MDL LANGUAGE

The Method Description Language (MDL) is aimed at describing the way of working of analysts when they perform database engineering activities. Firstly, we will recall the concepts on which the language is based. Secondly, we will give a short description of the language.

A. Basic concepts

The proposed design process modelling approach is based on the transformational paradigm according to which each *design process* transforms a (possibly empty) set of *products* into another set of products:

- a **product** is a document used, modified or produced during the life cycle of the information system; as we focus specifically on database specification, we will describe mainly database **schemas** and database-related **texts** such as DDL scripts, reports or application program sources.
- a **design process** is described by the operations that have been carried out to transform input products into output products; each operation is in turn a process; atomic processes are called *primitive processes*, while the others will be

called *engineering processes*; each process is supposed to be goal-driven, i.e., it tries to make its output products compliant with specific design criteria, generally called requirements [19];

- a **product type** describes the properties of a class of products that play a definite role in the system life cycle; a product is an instance of a product type;
- a **process type** describes the general properties of a class of processes that have the same purpose, and that process products of the same type; a process is an instance of a process type;
- the **strategy** of a process type specifies how any process of this type must be, or can be, carried out in order to solve the problems it is intended to, and to make it produce output products that satisfy its requirements; in particular, a strategy mentions what processes, in what order, are to be carried out, and following what reasoning. Only engineering process types are defined by a strategy. Primitive process types are basic types of operations that are performed by the analyst, or by a CASE tool.
- Several product types can be given the same, or similar, properties. Hence the concept of **product model**. A model defines a general class of products by stating the components they are allowed to include, the constraints that must be satisfied, and the names to be used to denote them. A product type is expressed into a product model¹
- A set of product types and process types define a **method**.

B. Short presentation of the MDL language

MDL is both a non-deterministic and a semi-procedural language. It is *non-deterministic* when it describes processes to be freely performed by humans. In this case, the CASE tool behaves as an assistant that suggests what to do and not how to do it. MDL can also be *semi-procedural* when it forces the analyst to do things in a definite order. In many cases, procedural specifications can be a natural approach to solve some classes of problems in a reliable way.

A method description is made of two parts. In the first one, product models are described. They are derived from the generic entity/object-relationship model (GER) described in [8] which is aimed at describing information structures as well as processing units. A product model is made of a subset of the GER concepts onto which constraints are specified. These concepts are those of the GER model which are renamed. For instance, an entity type is called a *table* in a relational model and an *object class* in an OO model. The constraints define the valid constructs of the specific model, as described in [24]. For instance, a relational model has no relationship types and all the tables have at least one column.

The second part of a method description concerns the process types. Like in most procedural languages, a process type as a signature and a body:

¹ For practical reasons we did not find it necessary to define the concept of process model, at least in a first step, thus making the architecture *inelegantly* asymmetrical. In particular, we identified a strong need for higher level abstraction above product types, while we found few convincing examples for process types.

- The signature is made of a name, and the list of product types they need as input, they produce in output or they can update.
- The product types of the signature are locally defined; they specify the product model they are derived from, as well as an identifying name and the number of instance that are allowed (exactly one, at least one, between two and five, no limit,...).
- The body is the strategy to be followed. This strategy is written in a traditional way, including some special instructions which are far less traditional. A strategy is a sequence of operations and control structures.
- An operation can be:
 - a call to another process type
 - a call to a built-in function of the supporting CASE tool
 - a call to an external function written in the built-in language of the CASE tool
 - a call to the use of a toolbox. A **toolbox** is a list of tools of the CASE tool the user can use at a given time. When the execution of a process of this type reaches a call to a toolbox, the method is suspended and the hand is passed to the user who is allowed to do whatever he wants in the CASE tool using only the allowed tools. This corresponds to an engineering activity that could not be formalised (in reverse-engineering for instance). The process goes on until the user explicitly declares his/her job is done.
- Control structures are the traditional *if...then...else, while, repeat*, as well as some special non-deterministic structures as *one* (choose one process in the given list and do it), *some* (choose one or several processes in the given list) and *each* (do all the processes in the given list, but in any order).

A more precise definition of the MDL language can be found in [25].

III. HISTORIES

The history of a database engineering process contains the trace of all the activities that were performed, all the products involved, all the hypotheses that were made, all the versions of the products resulting of those hypotheses as well as all the decisions taken. Naturally, the result is a complex graph. We will now examine this graph more precisely. But first of all, we will see why we need reusable histories. A more complete description of histories can be found in [26].

A. Reusable histories

An history can be reused in a great variety of ways, for different purposes, among which:

- the history can be used to make the database project evolve
- for documentation, it can simply be browsed or replayed
- the history of a reverse engineering part can be inverted in order to generate a possible forward engineering process that could have been followed at development time; this new history can be reused for reengineering [11]
- the history can be cleaned; all actions that do not participate directly to the development of the project can be removed;

this comprises processes performed according to some hypotheses that were rejected in later decisions, some simple tests (just to see what it would give), some actions followed by their inverse due to backtracking,... This cleaning can be useful in order to generate examples or tutorials to learn new analysts how to do

- it can be analysed in order to evaluate the quality of the work of the analyst
- it can be analysed in order to evaluate the method and to improve it.

B. History components

The first basic elements of histories are **processes**. A history should contain all the processes that are performed during an engineering activity that complies with a method. The method being specified in a procedural language, the resulting history is a tree of process calls. The whole project is made of processes, each of them being made of sub-processes and so on. Since a process is made of several sub-processes, it is useful to know in what order they have been performed, e.g., serially or in parallel. So each process will be stamped by its beginning date and time (mandatory) and end date and time. They will be identified by a name and the begin time stamp. In order to document his/her work, the analyst will add a description (some free text) to each process. We can have two kinds of processes: primitive processes at the operational level and engineering processes at the decisional level.

A **primitive process** is a process performed using only primitives (built-in function of the CASE tool or external functions written in the built-in language of the CASE tool). It can be performed by an analyst when the method allows him/her to use a toolbox or by the CASE tool itself when the method calls built-in or external functions directly. The execution of primitives can be recorded in a log file. Since the history is aimed at being reused, both by analysts and by the CASE tool itself, the log file has to be readable, precise and complete. A text file with a well defined syntax and the possibility to add comments and bookmarks seems to be a good solution. Since the primitives are database schema transformations and since these transformations can be formally defined and are proved to be reversible [13], precision and completeness are straight-forward if we store the signature of the transformations. In fact, we can store more than just the signature when needed because more information may be useful for reverse operations (e.g. it is necessary to keep the name of deleted attributes). Readability is achieved by the choice of a readable syntax for the transformation signatures and by the adjunction of comments.

An **engineering process** follows a strategy given by the method. As the analyst who follows the method can make hypotheses, try different solutions and decide to abandon some of them, it is no longer possible to record actions in a linear way like in the log file. The history of an engineering process is a graph. Hence, the whole history is in fact a tree of graphs; leaf nodes are primitive processes with their log files and non-leaf nodes are engineering processes with their graph.

Commonly, in software process modelling tools or business process modelling tools, engineering processes histories are well recorded, but they often use third party tools (editors, text processors, compilers, debuggers,...), the primitive processes in this paper, which have their own logging facilities, and all the logs are generally independent one from the other. In this paper, at the contrary, we link them all together in order to reuse them as a whole.

The second basic elements of an history are the **products**. Since an analyst has the possibility to generate different versions of a product, they will be identified by their name and their version. For the same reason as processes, we will add some descriptions to products. A given product can be the result of several processes, but, at some definite time, the product has to be declared as finished. From that moment, the product is locked. It is no longer possible to modify it. Hence, each product must have a locked/unlocked state. The type (schema or text) of the product has no influence on the history.

Finally, the third basic elements of an history are the **decisions**. A decision is a special kind of process, the sole difference being that it does not alter products, nor does it generate any product. There are two kinds of decisions. The first one is a decision that must be made according to the followed method. For instance, when the condition of an *if* or a *while* statement needs a response of the analyst (*if ask("Do you want to optimise the relational schema ?")...*). The second kind of decision is one that follows **hypotheses**. When an analyst has to perform a process, he/she can make different hypotheses and perform the same process several times with each hypothesis in mind. The description of each process will contain the hypothesis. Each process will generate its own **version** of the products. When all the processes are over, the analyst chooses one version among all to continue his/her work. The process of decision will show the choice and its description will contain the rationales that lead to that choice. This second kind of decision is not linked to the followed method, it can be made at any time.

IV. THE DB-MAIN CASE TOOL

After describing the main two components for process engineering activities, we will now see how we can link them in practice, with the use of a CASE tool. Since this work is performed in the DB-MAIN project, we will integrate processes to the DB-MAIN CASE tool. In a first time, we will describe briefly the CASE tool and see what are its particularities. Then, we will see how the repository has been extended. Finally, we will see how the human-machine interface of the CASE tool can be enriched in order to fulfil our goals: to bring a methodological assistance to the user and to record his/her work.

A. DB-MAIN, a generic CASE tool

DB-MAIN is a database engineering oriented CASE tool. A general software engineering CASE tool is more or less a control centre that allows a good integration of a series of third-party tools like general-purpose text editors, compilers,... A

database engineer also needs a series of tools to perform his/her job, but these are very specific tools like specialised graphical editing tools. All those specific tools are included in DB-MAIN.

Its main characteristics are the following:

- It uses a generic entity/object-relationship (GER) model for drawing database schemas. This model allows us to represent a very broad range of concepts from different abstraction levels and from different specific models; e.g. ER, NIAM or UML conceptual schemas can be drawn with the same generic concepts as for relational or COBOL logical schemas or for Oracle or Realia COBOL schemas at the physical level.
 - Transformation based : two semantically equivalent schemas can be mapped together by a series of reversible, semantics preserving transformations [9]. It allows us to smoothly transform a schema from an abstraction level to another one. All these transformations can be logged.
 - It manages complex history recording such as depicted in section III.
 - It is methodology-free. It is a space in which database engineers can work freely, performing all the standard and non-standard activities. It can be used as well for forward engineering, reverse engineering, evolution,...
- A more complete description can be found in [14] and [9].

B. Repository extension

For the history to be reusable, all its parts need to be correctly interconnected and attached to the followed method. [21] shows that necessity and how all the information are stored in his repository. In DB-MAIN, the repository will also contain all the information. But, since it is an integrated tool, all parts of the histories can be integrated to the repository, including the logs of the lower level tools. This allows us to be able to consider the history as a product that can, at its turn, be reused for different purposes, even transformed.

The DB-MAIN repository is a C++ object oriented database that stores a whole project, with all the products and the processes that are performed on them (see figure 1). Database schemas (GER schemas) are stored in a well structured way (via *sch_comp* relationship type), while texts are just a link to the files that contain them. A complete description of this repository can be found in [5].

The log of the primitive processes that modify products are attached to each product: each product has a *log* property which is a sequence of all the logs of the primitive processes separated with bookmarks.

As we saw in section III, engineering processes have much more complex histories: graphs. We can see on the lower half of figure 1 that an engineering process can be made of several sub-processes. Each process uses products in input, generates or updates products (usage field is respectively *input*, *output* or *update*). Hence, links between processes pass through products: if process A generates product P and process B uses P in input, we immediately see that A was performed before B; or, if process A and process B both use product P in input, we can say that the order in which they were performed does

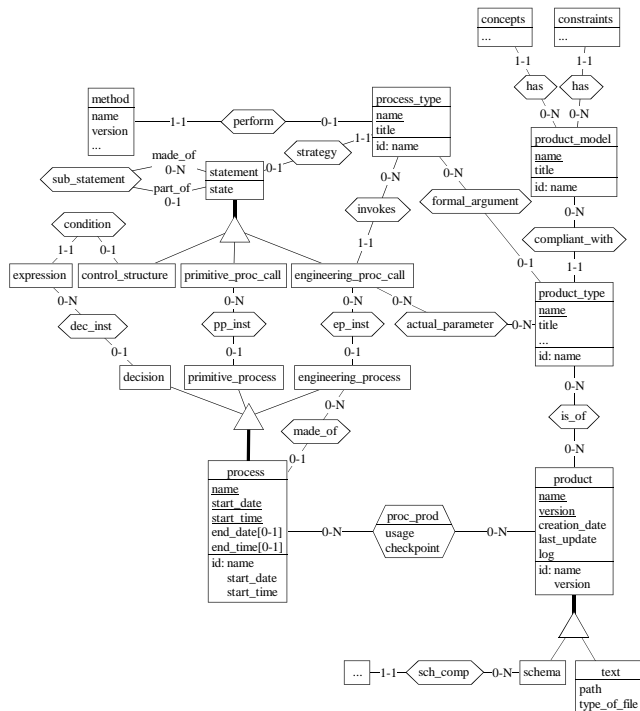


Fig. 1 - Part of the DB-MAIN repository concerning histories

not matter. Furthermore, the time and date fields of processes give more information about the real performance order.

The repository must also store the method that is followed during a project. A method written in the MDL language must be parsed before being used. The syntactical tree resulting of the parsing (LL(1) parser, see [1]) is stored in the repository. The top half part of figure 1 shows the C++ representation of this tree. We can see that the different processes and the different products of the history are linked (through *ep_inst*, *pp_inst*, *dec_inst*, *text_inst* and *sch_inst* relationship types) to their process types and product types in the syntactical tree.

With such a repository, it is easy to present the syntactical tree of the method in a graphical algorithmic way (see [24]) and the history in various ways (see [25]).

C. Human-machine interface extension

DB-MAIN, in its original form, is a generic CASE tool which allows a database engineer to do whatever he or she wants, but without any methodological guidance. Now that we have added a method in the repository, we will examine how we can extend the CASE tool environment in order to constrain and help the engineer. All along the way, we will keep in mind that if the engineer needs to be constrained, he/she also needs some freedom. A CASE tool that would not let any freedom is bound to be rapidly abandoned.

In this section, we will see how we can present the method to be followed by the engineer, then how he/she is helped and constrained to follow it. We will also see that we can modify the look and feel of the CASE tool according to the method. Finally, we will explain how the methodological engine can check the engineer's work.

i. Method presentation

Most method description languages on the market are declarative languages. For each action that can be performed, conditions of enactment have to be specified. At a given time, according to the current state of the products in development, the conditions can be evaluated and the processes that can be enacted can easily be listed. So, it is easy for an engineer to know what to do at that time. But these tools lack the possibility to show globally to the engineer the way of working he has to follow. He/she can know what to do now, but not what to do next. It is like wandering in a labyrinth with very high walls all around you. You know the way you have followed up until now, you know your goal is to go out of there, but you do not know what still remains to be explored. If the labyrinth is on a paper and you can see it all in once, it is much easier to find the way out of it. That is what we will do, we will show it all in once.

The MDL language is procedural. Like every procedural language, it can easily be shown in an algorithmic way: process types are drawn as rectangles and the control flow is shown with arrows. Figure 2a shows a small example of an engineering process type where the control flow starts at the first half circle (round line at the bottom), follows bold lines, and ends at the second half circle. Conditions (in *if...then...else* structures or loops) are drawn as diamonds. The main difference with traditional procedural language is the non-determinism: loops can have no condition, or some special structures forces the engineer to take himself/herself the decision of the path to follow. On figure 2, we can see a *some* structure (the user can choose to perform processes of one or many of the sub-process types, in the order he/she wants) enclosed in a loop without condition.

The product types that are used in input, output or update of the process types are shown with ellipses and their usage (the data flow) with thin arrows.

Engineering sub-process types can themselves be drawn the same way. An engineer can know how to perform a *COBOL schema extraction* by double-clicking on its rectangle in the figure 2a to draw the algorithm of its strategy, which is shown on figure 2b. Primitive processes have no strategy, but a more detailed description (a textual description, the list of the tools that can be used,...) can be obtained as well.

ii. The use of the method

To show the method to follow is interesting in itself, but one of our main goal is to help an engineer to follow it. This help must have the following characteristics:

- it must be clear, a genuine reference, that can answer all the questions an engineer can have
- it must be able to say to the engineer, at every moment, what he/she can do
- it must constrain the engineer to do precise things
- at the same time, it must allow the engineer to have some freedom to perform his job at his/her will; for instance, to draw a schema in which every entity type must have at least one attribute, the methodological engine should not enforce the engineer to add an attribute straight away, what matters

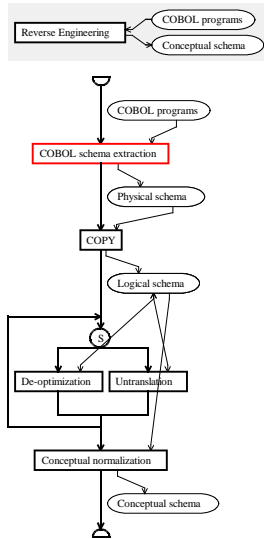


Fig. 2a A process...

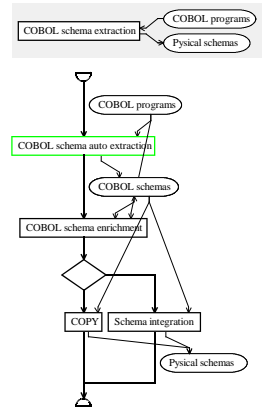


Fig. 2b ...and one of its sub-processes

is just that every entity type has at least one attribute when the whole schema is finished

- it can automate some work in order to relieve the engineer from tedious repetitive actions, asking him/her to intervene only when necessary, when a decision has to be taken.

The first point seems to be obvious. Every good book about a method is in accordance with that. The way we show the method, presented here above, is a good starting point. By adding to every process type and to every product type an explanation in natural language we can complement and clarify these algorithms in order to achieve this goal. These explanations can be shown on demand. That way, the algorithm is really a full hypertext documentation of the method. This is more difficult to achieve with declarative languages.

The second point, on the other hand, is so basic that every CASE tool can do it. We add a state to each process type in the syntactical tree of the method (see figure 1). This state will be shown on screen with colours. Process types can have four possible state value: *waiting*, *executable*, *in progress* and *done*. In figure 2a we can see that the first process type is drawn in a lighter colour than the others, it is in the *in progress* state. On figure 2b, the first process type is in still another colour, it is in the *executable* state. All others are in the *wait-*

ing state. It shows the engineer that he/she can perform a process of that type. When he/she finishes that process, the colours are updated. In figure 2a, when the engineer finishes a process of the second type (*COPY*), since the control flow indicates a *some* structure, two process types (*De-optimization* and *Untranslation*) will be coloured (get the *executable* state). So, the engineer will be able to choose between both of them.

At the finest grain of the decomposition, we find primitive process types. Some of them are a simple use of a built-in tool (like the *COPY* process). When this use is over, the process type state can be set to *done* automatically by the methodological engine. Other primitive process types allow the engineer to work by himself/herself (for instance, he/she is allowed to draw a schema). The methodological engine waits for him/her to say he/she has finished: the *done* state has to be set manually. An engineering process can be declared *done* both automatically or manually as we will see later.

As we said in the fourth characteristic, people like freedom, and it is necessary to allow an engineer to do some special things. One of them² is the possibility to do the same thing several times, starting by stating different hypotheses. A process type which already has instances can be asked to be performed one more time. This will create a new branch in the graph of the history, stamped with the new hypothesis. From there, the first colour will be used again. Thus, both colours can be found at several places in the hypertext algorithm.

If several process types are in the *executable* state, the engineer has to choose which one to perform. But, if only a process of one type can be executed, the method engine can do it automatically. Hence, the method engine can do a large part of the job and stop to ask the engineer to intervene only when necessary, either because an engineering process type contains a non-deterministic structure (which makes at least two processes to be coloured), because a primitive process type is encountered that needs to be performed by an engineer, or because a decision needs to be taken. So, the methodological engine can start processes, follow their strategy as long as it does not encounter decisions to be taken, execute built-in primitive processes and terminate engineering process automatically. But, sometimes, engineers like to know what is going on so it is necessary for that automation to be turned off.

Since the syntactical tree of the method and the history are stored in the same repository and since the engineer enacts processes by selecting coloured boxes on the algorithmic representation of the syntactical tree, the recording of the history is straightforward.

iii. Toolboxes

Some primitive processes can only be performed by a human engineer, using a toolbox. We already saw that the methodological engine can let the engineer work by his/her own and wait or him/her to declare that the job is done.

In the MDL method definition, the method engineer can define toolboxes (see section II). For instance, to edit a relational schema, the method engineer could define a toolbox with tools for creating, deleting or editing tables, columns,

²We will not examine all possible liberties in this paper, it should be rather long, and of few interest here.

primary keys and foreign keys. When the method needs for the database engineer to work, it has to specify with what toolbox. When the methodological engine encounters this primitive process, it has to adapt the CASE tool in order to allow the engineer to use the tools in the toolbox and to disallow the use of all other tools. Hence, the methodological engine has to add or remove entries from the menus, enable or disable buttons in dialogue boxes, enable or disable keyboard shortcuts and monitor mouse clicks in the schema views.

iv. *Lexical adaptation of dialogues*

A product model, as we showed in section II, is made of two parts: concepts and constraints.

The methodological engine must use the concepts to adapt the CASE tool interface to the model of the product the engineer works on. For instance, if a relational schema model defines the concepts of, among others, table (renaming of entity type in the GER), and column (attribute in the GER), then the CASE tool interface must be updated by replacing every occurrence of the words *entity type* and *attribute* by the words *table* and *column* respectively, each time the engineer works with a relational model compliant product. This implies to keep the menus, the dialogue boxes and all the messages up-to-date.

v. *Automatic checking*

The constraints defined in every product model have to be used to insure that the engineer respects the rules of the game when allowed to use a toolbox. But, as we said above, we do not intend to refrain the engineer to work as he/she likes. For instance, a relational model may demand that all tables have at least one column with the constraint:

ATT_per_ET (1,N)

stating that the number of attribute (renamed *column* by the concepts) per entity type (*table*) must be comprised between 1 and N. A possible solution should be to check if that constraint is verified each time the engineer performs an action and react if it does not; when a table is created, the methodological engine may react by forcing the creation of a column, or it could impeach the removal of the last column of a table. But this can be against the engineer's will. The fact that the schema is really compliant to the relational model is important when the engineer states the job is over, but, during his/her work, a temporarily non-compliant schema is perfectly acceptable. The engineer can prefer to create several tables at once, then add the columns. Hence, a better solution is to let the engineer work freely with all the tools of the current toolbox and validate the schema when he/she wants to change the state of the process type to *done*. If the validation fails, the state is not changed and a list of the violated constraints with the elements of the schema that violate the constraints is reported. This is an hypertext list, a click on an element in the list selects that element in the schema.

In some occasions, it can be useful to give still more freedom to the engineer: it is preferable if the schema is perfectly compliant to its model, but some transgressions can be allowed. In that case, the method engineer can define, with the MDL language, a weakly compliant product type. When the

methodological engine encounters such a situation, the automatic validation is performed and the list of validation is reported, but a message box informs the engineer he/she can go on anyway, informs him/her of the risks to do so and asks him/her what to do.

It is to be noted that this validation of the schema that is automatically performed at the end of a job can also be manually requested by the engineer as an help.

V. CONCLUSION

DB-MAIN is originally a generic database oriented CASE tool, but we saw, all along this paper, how it can be methodologically personalised. It allows an engineer to be guided and constrained to respect the rules of the game, while relieved from tedious work and free to act and take decisions as often as possible.

An evaluation version of the DB-MAIN CASE tool is available on the following site:

<http://www.info.fundp.ac.bel/~dbm>

REFERENCES

- [1] A. Aho, R. Sethi, J. Ullman, *Compilers: principles, techniques and tools*, Addison-Wesley Publishing Company, 1985
- [2] B. Curtis, M. I. Kelner, J. Over, *Process Modeling*, Communications of the ACM, September 1992, Vol.35 No.9, pp. 75-90.
- [3] Gregor Engels and Luuk P.J. Groenewegen, *SOCCA: Specifications of Coordinated and Cooperative Activities*, in A.Finkelstein, J.Kramer, and B.A. Nuseibeh, editors, *Software Process Modelling and Technology*, pp. 71-102. Research Studies Press, Taunton, 1994.
- [4] J. Ebert, R. Süttenbach, I. Uhe, *Meta-CASE world-wide*. Technical report 24/98, Fachberichte informatik, Universität Koblenz-Landau, Institut für Informatik, Rheinland, Koblenz, 1998.
- [5] V. Englebert, *Voyager 2 reference manual*, technical DB-MAIN documentation.
- [6] Jacky Estublier and Rubby Casallas, *The Adele Configuration Manager*, in Tichy editor, *Configuration Management*. John Wiley & Sons, 1994.
- [7] G. Grosz, S. Si-Said, C. Rolland, *Mentor : un environnement pour l'ingénierie des méthodes et des besoins*, Actes du Congrès INFORSID, pp. 33-51, Bordeaux, juin 1996.
- [8] J-L. Hainaut, *A Generic Entity-Relationship Model*, in Proc. of the IFIP WG 8.1 Conf. on Information System Concepts : an in-depth analysis, North-Holland, 1989.
- [9] J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland, *Database evolution : the DB-MAIN Approach*, in Proc. of the 13th International Conference on ER Approach, Manchester, Springer-Verlag, LNCS 881, 1994

- [10] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, D. Roland, *Requirements for Information System Reverse Engineering Support*, in Proc. of the IEEE Working Conference on Reverse Engineering, Toronto, IEEE Computer Society Press, July 1995
- [11] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, D. Roland, *Database Reverse Engineering : from Requirement to CARE tools*, Journal of Automated Software Engineering, 3(2), 1996, Kluwer Academic Press.
- [12] J.-L. Hainaut, J. Henrard, J.-M. Hick, D. Roland, V. Englebert, *Database Design Recovery*, in Proc of the 8th Conf. on Advanced Information Systems Engineering (CAISE'96), Springer-Verlag, 1996.
- [13] J.-L. Hainaut, *Specification preservation in schema transformations - Application to semantics and statistics*, Data & Knowledge Engineering, 16(1), 1996, Elsevier Science Publish.
- [14] J. Henrard, V. Englebert, J.-M. Hick., D. Roland, J.-L. Hainaut, *DB-MAIN: un atelier d'ingénierie de bases de données*, in Ingénierie des Systèmes d'Information, Vol. 4, n° 1/1996, pp. 87-116.
- [15] J. Henrard, J.-M. Hick, D. Roland, V. Englebert, J.-L. Hainaut, *Techniques d'analyse de programmes pour la rétro-ingénierie de base de données*, Actes du congrès INFORSID, pp. 215-232, Bordeaux, 4-7 juin 1996.
- [16] M. Jarke, editor. *Database Application Engineering with DAIDA*, Springer - Verlag, 1993.
- [17] S. Kelly, K. Lyytinen, M. Rossi. *MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, Proceedings of the 8th International Conference CaiSE'96 on Advanced Information Systems Engineering, LNCS 1080, pp. 1-21, Heraklion, Crete, Greece, May 1996, Springer-Verlag.
- [18] R. J. Mayer, P. C. Benjamin, B. E. Caraway and M. K. Painter, *A Framework and a Suite of Methods for Business Process Reengineering*, www.idef.com/articles, October 30, 1998.
- [19] J. Mylopoulos, L. Chung, B. Nixon, *Representing and Using Nonfunctional Requirements: A Process-Oriented Approach*, IEEE TSE, Vol. 18, No. 6, June 1992, pp. 483-497.
- [20] Nature Team, *Defining Visions In Context: Models, Processes And Tools For Requirements Engineering*, Information Systems, Vol. 21, No 6, 1996, pp. 515-547.
- [21] K. Pohl, *Process-Centered Requirements Engineering*, Research Studies Press Ltd, 1996
- [22] K. Pohl, K. Weidenhaupt, R. Dömges, P. Haumer, M. Jarke, R. Klamma, *PRIME: Towards process-integrated environments*, to appear in ACM Transactions on Software Engineering and Methodology.
- [23] C. Potts, G. Bruns, *Recording the Reasons for Design Decisions*, in ICSE 88, 1988, pp. 418-427.
- [24] D. Roland, *Un langage de description de processus : définition des prédicats structurels*, technical report, septembre 1995.
- [25] D. Roland, J.-L. Hainaut, *Database Engineering Process Modelling*, Proceedings of the first International Workshop on the Many Facets of Process Engineering, pp. 37-49, Gammarth, Tunisia, September 22-23, 1997.
- [26] D. Roland, J.-L. Hainaut, J. Henrard, J.-M. Hick, V. Englebert, *Database engineering process history*, Proceedings of the second International Workshop on the Many Facets of Process Engineering, pp. 63-76, Gammarth, Tunisia, May 1999.
- [27] J. Souquières, N. Lévy, *Description of Specification Developments*, in Proceedings of RE'93, San Diego (CA), 1993.
- [28] X. Wang, P. Loucopoulos, *The Development of Phedias: a CASE Shell*, Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, pp. 122-131, Toronto, July 10-14, 1995.