DSTM: a framework to operationalize and refine a problem solving method modelled in terms of tasks and methods

Francky Trichet & Pierre Tchounikine

{Francky.Trichet, Pierre.Tchounikine}@irin.univ-nantes.fr
IRIN, Université de Nantes & Ecole Centrale de Nantes
2, rue de la Houssinière, BP 92208 44322 Nantes cedex 03, France

Abstract

In this paper we present DSTM, a framework that enables the operationalization and the refinement of problem-solving methods modelled within the Task–Method paradigm. DSTM proposes an operational kernel, i.e. operational but flexible high-level constructions: modelling primitives, such as task or method, and manipulation mechanisms, such as select a method. These constructions can be customized in order to better capture the paper-based model to be operationalized. This permits the construction of an implemented system that is an explicit reification of the paper-based model, and, therefore, enables to analyze the model by means of the analysis of the system. In order to support this analysis, DSTM proposes tools that allow the knowledge-engineers to inspect the implemented model from different points of view. This facilitates the conciliation of the initial modelling phase, the model refinement phase and the operationalization phase that are achieved when constructing a KBS.

Keywords: Problem-solving method; Task–Method paradigm; Paper-based model

1. Introduction

The modern view (David et al., 1993) of the construction of knowledge based systems (KBS) is a four step process. First, some expert knowledge is informally acquired. Then, this knowledge is analyzed by the knowledge-engineers in order to construct a conceptual model of the problem-solving method (PSM) to be used. This model is an abstract implementation-independent description of the problem-solving process that the system must apply, in terms of tasks (what is to be achieved) and methods (what means can be used to achieve these tasks). Top-level tasks are associated with one or several methods (called “decomposition methods”) that decompose them into subtasks, recursively, until these subtasks are sufficiently simple to be directly tackled by “execution methods”. Once this “skeletal” was defined, it could be used to instantiate the model. This phase consists in using the conceptual model as a guide to acquire the domain expert knowledge (i.e. the knowledge that is necessary to perform the execution methods) that was not identified in the initial phase. Finally, the obtained instantiated model is operationalized in order to construct an executable system.

Because the elaboration of the conceptual model is indeed the key-point of this process, most works focus on this phase, and the operationalization of this model is generally considered as a simple “technical” phase. Therefore, the modelling phase and the operationalization phase are completely dissociated. However, when constructing effective KBSs, paying some attention to the operationalization phase appears more important, as it is not as outside of the “conceptual work” that it is often argued. Problems missed during the elaboration of the model often only appear when operationalizing and testing the KBS (see for instance Linster, 1993 or Vanwelkenhuysen and Rademakers, 1990). Very often, the operationalization phase puts into evidence problems that can easily be tackled, such as punctual gaps or contradictions in the domain knowledge that affects how a particular leave-task is achieved (i.e. problems related to the model instantiation). More worrying, the operationalization phase can also put into evidence that the PSM itself is ill-defined, i.e. put into evidence that some tasks or methods, as they are currently defined, do not permit the problem-solving competence that was expected. Such problems address modelling issues, and require the knowledge-engineers to analyze and refine the initial paper-based conceptual model.

Modelling problems must be discovered and tackled before the time consuming phases of acquiring all the domain knowledge and implementing the complete system. This can be achieved by operationalizing the elaborated paper-based model with a subset of the domain knowledge, analyzing if the way the tasks and methods are defined allows emulating the expected problem-solving behaviour and, if necessary, refining the model. However, modelling problems must be tackled by knowledge-engineers at a model level, and not at the implementation level, and, therefore, the operationalization approach must allow the knowledge-engineers to use the implemented system as a means to evaluate and refine the PSM model. Current operationalization languages (i.e. languages dedicated to the operationalization of PSM paper-based models) propose high-level constructions (e.g. Task and Method modelling primitives and mechanisms to manipulate these primitives) that facilitate the operationalization phase, but are not satisfactory in order to use the implemented system as a means to refine the model. First, they impose generic built-in constructions (i.e. implicitly, a modelling language) that constrain how the PSM can be expressed. When the knowledge-engineers use domain related notions to describe the model components, the fact that the operationalization language only proposes “generic” built-in unadaptable notions introduces a distortion between the paper-based description of the PSM and its operational description, this distortion limiting the capacity to study the PSM properties through its operational description. Second, current operationalization languages provide almost no support to the knowledge-engineers to analyze if the PSM that is being implemented allows emulating the expected behaviour.

In order to enable using the operationalization phase as a means to evaluate and refine the PSM, we propose to use operationalization environments based on flexible operational kernels. An operational kernel proposes predefined but flexible high-level constructions (e.g. modelling primitives such as Task or Method and manipulation mechanisms such as *Select a Method*). It provides more flexibility than an operationalization language because these constructions (and, therefore, the modelling language they define) can be customized in order to better capture the paper-based model. This enables the construction of an implemented system that is an explicit reification of the paper-based model, and, therefore, enables to analyze the model by means of the analysis of the system. In order to support this analysis, we propose to construct tools that allow the knowledge-engineers to inspect the implemented model from different points of view. In summary, we propose to achieve the operationalization phase with a framework that allows both modelling features (definition of an adapted modelling language, analysis and refinement of the PSM model) and operationalizing features (making the model operational). We believe such a framework is more prone to conciliate the initial modelling phase, the model refinement phase and the operationalization phase that are achieved when constructing a KBS.

In this paper we present DSTM, the modelling and operationalizing framework we have constructed to enable the operationalization and the refinement of PSMs modelled within the Task–Method paradigm. In Section 2 we present an example of a Task–Method paper based model, why current approaches of the operationalization phase are not satisfactory to operationalize and refine such a model and the principles that underlie DSTM framework. In Section 3 we present how DSTM allows operationalizing such a model and what tools are currently available to help the knowledge-engineers in their analysis of the implemented system. In Section 4 we discuss the scope of this approach (highlighting it is useful both when the PSM model is abstracted from rough expert knowledge and when one reuses a predefined generic model), we compare it to related works and we present the current direction of the work. We conclude in Section 5.

2. Operationalizing problem-solving methods

2.1. An example: operationalizing and refining Emma PSM

Emma[1] (Choquet et al., 1997a, 1997b) is an educational system (under construction) that aims at training students in the practice of linear programming as a technique to solve concrete problems (for example economic problems). In order to be able to solve problems and to analyze students' resolutions, Emma embodies a KBS. This KBS implements a problem-solving method whose conceptual model was elaborated with the maths teacher, on the basis of concrete problem resolutions (thus, the model was constructed by abstraction from the teacher's knowledge).

In the first stages of the modelling process, Task and Method notions were used as informal modelling guides. Analyzing the teacher's problem-solving and discussing with him with the objective to identify aspects of the solving that can be dissociated (tasks) and means that can be used to tackle these different aspects (methods) helps in abstracting from rough knowledge. The details of how a method is effectively achieved can be tackled later on and do not interfere with the modelling of the general problem-solving behaviour. In Emma, this led to the identification of high-level tasks such as “Formalisation of the mathematical situation”, which are then decomposed into subtasks such as “Variable definition” or “Definition of the type of objective function”, and methods to achieve these tasks (for instance, the task “Solution of a linear programming problem” is associated with multiple methods such as “Method of tables” or “Graphic method”).

When the analysis becomes precise enough, the tasks and methods characteristics that must be denoted and how these characteristics will be used to select the pertinent tasks and methods during resolution must be defined. Different modelling aspects that were domain related (i.e. specific to Emma domain and Emma context of use) appeared when the domain tasks and methods were defined more precisely. As a simple, but intuitive example, in Emma, the pertinence of a task is considered from two different points of view: the mathematical point of view and the taught-method point of view. An action can be mathematically possible, but not pertinent to the taught-method (for instance because teachers want the students to

[1] Because it will permit us to present examples from the different ideas presented in this paper, we will use Emma PSM model as an illustration in the remaining of this paper. However, it should be noted that the scope of our work is not limited to modeling problem-solving in educational systems.

consider some other aspects before) or can be in accordance with the taught-method (i.e. point at a pertinent objective), but mathematically impossible. The objective of this modelling phase is, therefore, not only to identify all the tasks and methods. It also aims at the identification of the notions that are important to describe the PSM, i.e. the identification of an adapted modelling language. Identifying such an adapted modelling language is very important as it defines guidelines for interviewing the experts and acquiring the rest of the expertise (what information must be acquired for every task and every method is known). When this process is stabilized, an incomplete "skeletal" paper-based model is obtained. How tasks and methods should be denoted (i.e. the modelling language) and the major (if not all) tasks and methods (i.e. the PSM structure) are identified. Instantiating this model consists in defining the remaining tasks and methods (if any) and the domain knowledge, in a process guided by the skeletal model. Operationalizing this model consists in representing the tasks and methods in the operational framework (i.e. translating the paper-based description of the tasks and methods into the operational version of the modelling primitives, in order to obtain the Task–Method knowledge-base), implementing the selection mechanisms (that will manipulate the Task–Method knowledge-base) and the execution methods (manipulation of the domain knowledge).

2.2. Operationalization languages and their drawbacks

An operationalization language is a language that enables making a conceptual model operational. Examples of operationalization languages are Omos (Linster, 1993), Model-K (Karbach and Voss, 1993), Karl (Fensel, 1995), Aide (Greboval and Kassel, 1992), Task (Talon and Pierret-Golbreich, 1996), Lisa (Jacob-Delouis and Krivine, 1995), MML, (Guerrero-Rojo, 1995) or Expect (Gil and Paris, 1996). The ancestors of operationalization languages are the expert-systems shells (e.g. Emycin), which proposed low-level formalisms such as production rules. An intrinsic limitation of using low-level formalisms is that the organization of knowledge that is defined while modelling is lost in the implementation. All recent works agree on the fact that it is necessary to use higher-level languages, that allow preserving a structural correspondence principle between the model and its implementation [i.e. to every component of the model corresponds an explicit component in the implementation (Reinders et al., 1991)]. The respect of this principle is known as facilitating the maintenance and the evolution of the KBS, "knowledge level" reflective control (Reinders and Bredeweg, 1992) and the construction of explanations to the end-user (Greboval and Kassel, 1994), i.e. using the operational system as a means to achieve model-based activities.

An operationalization language proposes a set of modelling primitives and inference mechanisms that can manipulate knowledge represented as instances of these primitives. A certain number of languages propose primitives that correspond to that of their underlying methodology. For instance, languages such as Omos, Model-K or Karl propose structures that correspond to the primitives that are used in the Kads methodology (Wielinga et al., 1992). Some other languages propose constructions proposed as "epistemological primitives". Typically, most languages (e.g. Aide, Task or MML) propose Task and Method primitives, that allow representing control knowledge in a way that denotes Newell's rationality principle (Newel, 1982). A set of selection mechanisms allow a simple hierarchical control (predefined decomposition of tasks in sub-tasks) and/or an opportunistic behaviour (selection at run time, according to the problem-solving context, of the more pertinent task to consider, and, then, of the more pertinent method to achieve it). Associated to these Task and Method primitives, a lower-level representation language is used to describe the domain knowledge and the inferences that can be achieved on this knowledge. Domain tasks and methods are defined by filling the slots proposed by the modelling primitives with knowledge related to the domain and/or to the problem currently under study.

Problem 1: *current operationalization languages impose their modelling point-of-view*.

Current operationalization languages impose a predefined modelling language, the one that underlies their hard-encoded operational constructions. For instance, current Task–Method languages impose a particular definition of a Task (i.e. what notions are to be denoted when describing a Task or, from a technical point of view, what are the slots to be filled, e.g. *objective*, *preconditions* or *resources*), and impose a particular definition of how tasks of the knowledge-base are selected at runtime (i.e. what are the criteria that are considered when selecting a task or, from a technical point of view, what slots are considered and what is the "formula" that defines if the task is to be selected). Adopting particular definitions facilitates the implementation of the language, whose mechanisms can be hard-encoded. However, from a modelling point of view, the consequence is that if the paper-based model does not smoothly fit into the proposed modelling language, it must be translated into these structures, and the correspondence model/implementation is lost. For instance, in Emma model, the pertinence of selecting a task is denoted by two different notions (mathematical point of view and taught-method point of view). Such a dissociation greatly clarifies the analysis of the rough knowledge that is obtained from the teacher. Existing Task–Method languages only propose a "generic" *preconditions* slot. Emma two criteria could be compiled in this single slot. This would not affect the problem-solving behaviour of the system. Nevertheless, this would introduce a distortion between the model and the implementation that would heavily[2] limit how the system can be used for model-based activities. In particular, if the problem-solving behaviour is not satisfactory, interpreting this behaviour at the model level will be made more difficult. Mixing different modelling notions will also limit the explanations that can be provided to the end-used (and, in the case of an educational system such as Emma, the teaching capacities would also be affected).

Problem 2: *operationalization languages propose no help for refining the PSM model*.

In most operationalization languages, the only support that is provided in order to help the knowledge-engineers to validate (and eventually refine) the model is limited to what derives from the fact the PSM can be run: tracing facilities and/or asking why or why not questions (van Heijst et al., 1997). There is no attempt to provide the knowledge-engineers with more synthetic information, that could help them in discovering ill-defined parts of the model. For instance, in a non trivial model such as Emma, a task can be achieved by several methods, and achieving a task with one or another method can have some influence on the rest of the solving (the methods can produce different results, and, therefore, influence the selection of the following tasks and/or methods). Keeping aware of this influence is not trivial, it requires the knowledge-engineers to keep in mind a synthetic understanding of how complex objects interact one with another. However, because they consider the operationalization phase as being essentially a coding phase, current operationalization languages do not aim at supporting the knowledge-engineers in this task. Expect and Omos propose some help to acquire domain knowledge (analysis of what domain knowledge is missing in the system), but not to analyze the overall problem-solving competence modelled in the system.

Problem 1 is not to be considered if the modelling of the PSM is made according to the modelling primitives proposed by the target operationalization language. Our opinion is that proposing guidelines (e.g. Task and Method notions) to the knowledge-engineers in order to help them structuring rough knowledge is useful, but imposing unadaptable structures is too

[2] Of course, such a single (and simple) distortion would remain tractable, enough to allow a second one and so on, until the knowledge-base is no longer understandable by anyone.

constraining. In other terms, we think the precise definitions of what is a Task and what is a Method, that defines what modelling notions are to be considered, must be defined according to the expertise to model and the context of use of the KBS, and the operationalization language must follow, and not the contrary.

Problem 2 is not to be considered if one supposes the paper-based model as definitely correct. Our opinion is that although the general organization of the PSM must be defined before any implementation work, necessary refinements of (parts of) the PSM often appear when confronting the paper-based model with what knowledge can be acquired from experts (i.e. when instantiating the model) and how concrete problems can be solved (i.e. when testing the system). Therefore, it is worthwhile to adopt an implementation approach that will facilitate these refinements. For this purpose, we propose to operationalize the incomplete "skeletal" paper-based model, with a strict adherence to the model in the implementation, and to use this operationalization process as a means to analyze, refine (if necessary) and instantiate the model.

3. The DSTM framework

DSTM allows tackling problem 1 by proposing an operational kernel, i.e. customizable operational structures for the Task and Method modelling primitives and their selection mechanisms. The default structures can be directly used; in such a case, the operational kernel is used as yet another Task–Method operationalization language. If the default structures do not allow a satisfactory representation of some specificities of the paper-based model, they can be modified. DSTM thus allows the construction of an operational system that respects a strict structural correspondence with the model (the system reifies the model).

DSTM allows tackling problem 2 by enabling the construction of tools that can analyze the implemented system (and, therefore, the model it reifies). This is made possible by the fact that the operational version of the knowledge-base (the tasks, methods and domain knowledge) and the manipulation mechanisms are represented explicitly, and not compiled as blackboxes. At present, two main types of tools have been constructed. The first type is concerned with checking if the Task–Method knowledge-base respects some constraints. Two types of constraints can be checked. First, constraints defined, explicitly, with a graphical Entity-Relationship language (e.g. "each task must be associated with at least one method"). Second, the constraints that are defined, implicitly, by the selection mechanisms (i.e. detecting that one of the selection mechanism uses the slot S_i of the Method modelling primitive as a criteria and, therefore, defining a method without valuating the slot S_i can be problematic). The second type of tools is concerned with presenting to the knowledge-engineers a synthetic view of how, given the current selection mechanisms, the tasks and methods of the knowledge-base can interact, i.e. presenting a synthetic view of the problem-solving competence that is modelled in the implemented system. A common characteristic of these tools is that they are metatools in the sense introduced in (Eriksson and Musen, 1993): they are based on a reflective analysis of the implemented systern. Therefore, when the operational kernel is modified, these tools adapt themselves, and can be used without further programming work. Moreover, this "meta" approach will allow us to develop our framework by integrating new tools without modifying the rest of the framework.

Section 3.1 presents how DSTM operational kernel can be used to operationalize a model, and Section 3.2 presents what help can be provided by current DSTM tools. The model that is used as an illustration is Emma. The different figures are snapshots from the current tentative graphical interface.

3.1. Operationalizing a Task–Method model with DSTM

DSTM operational kernel is structured in four layers. The top level (level 1) is a general control[3] of high-level actions such as *Select a Method*. These high-level actions (level 2) are based on abstract notions such as the *applicable Method* notion; an abstract notion (level 3) denotes a possible state of a Task or a Method during resolution. The final level (level 4) is the description of the Task and Method modelling primitives. The kernel proposes a default definition of the modelling primitives, that induces default definitions for the abstract notions and the high-level actions [these definitions are inspired from related works, in particular (Jacob-Delouis and Krivine, 1995)]. If these definitions do not allow denoting the paper-based model in a satisfactory way, the modelling primitives, the abstract notions, the high-level actions and the general control can be adapted. This is made possible by the fact that the definitions that are adopted for these different items are explicitly denoted in the source-code, and can easily be modified. The implementation respects a limited-interaction principle, in order to limit the propagation of the modifications, if any.

As said before, operationalizing a PSM model also requires modelling the basic domain knowledge (level 0). For this purpose, DSTM framework proposes a relational language and an Entity-Association language. Presenting these languages in details is outside the scope of this paper (see Trichet, 1997) for such a presentation), and we will just mention how Emma domain knowledge was modelled with the relational language. We then present, successively, the modelling primitives, the abstract notions, the high-level actions and the control levels (the default definitions and how they were adapted according to Emma model).

The domain knowledge

The *Yearl* (Trichet, 1997) relational language allows representing domain knowledge by defining types of facts, each of these types being associated with a set of possible values. A domain fact is represented as a couple (object, value). For instance, in Emma, three different types of facts were dissociated. Concrete domain facts, that denote information on the solution, e.g. (there is) "one explicit constraint" or "number variables=2", can be true or false. Abstract domain facts, that denote information on the solving state that is related to mathematical notions, e.g. (it is an) "optimization problem with constraints" or (it is a) "linear programming problem", can be impossible, unlikely, possible, likely or certain. Strategic facts, that denote information on the solving state that is related to the taught-method, e.g. (the) "problem type", can be defined or undefined. Facts can be connected using relations such *as is-a-particular-case*, *is-a-possible-value*, *are-exclusive-values*, *imply* or *are-equivalent*. What is obtained is a graph such as the one presented in Fig. 1. Every relation is associated with the inference mechanisms that correspond to its semantics. While problem-solving, the concrete facts values are identified, and the inference mechanisms are activated in order to propagate the values.

[3] We will focus here on the control that allows a dynamic selection of tasks and methods, i.e. a context where at a state of the resolution different tasks can be considered and one of these is selected, at runtime, according to the current context and then, one of the different methods that exist for this task is selected, here again at runtime. This is the more general context, cases such as static hierarchical decomposition of tasks or existence of a single method for a task are variations.

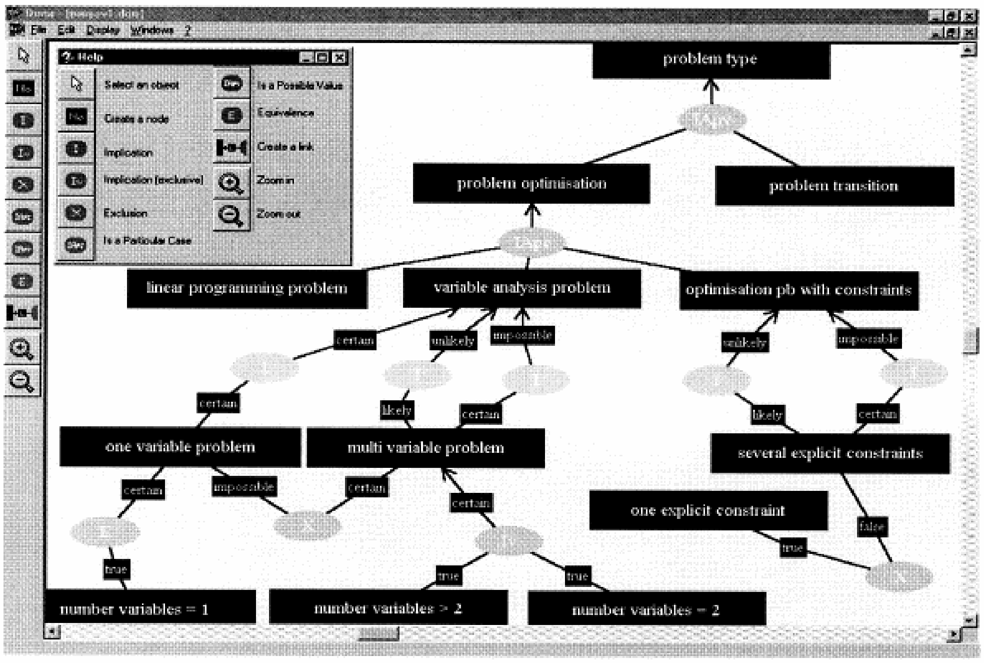


Figure 1. The domain graph for the Emma system (extract).

The modelling primitives

DSTM predefined primitives are Task and Method. In the default definitions, a Task is defined (cf. Fig. 2) by its results and the context in which it can be achieved. If known, methods that can achieve it can be associated. A Method (cf. Fig. 3) is defined by the results it produces, the context in which it can be fired and the knowledge required for its achievement. If known, the description of when it is particularly relevant can be added. A method can directly carry out a task or split a task into subtasks.

Slots	Definitions
Name	*Name of the Task*
Expected Results	*Description of the expected results*
Input Context	*Description of the context in which the Task can be achieved*
Associated Methods	*If known, a list of Methods that can achieve the Task*

Figure 2. Description of a Task.

Slots	Definitions
Name	*Name of the Method*
Results	*Description of the produced results*
Input Context	*Description of the context in which the Method can be fired*
Requirements	*Description of what is required for the execution of the Method*
Favourable Context	*Description of the context in which the Method is particularly relevant*
Decomposition	*Explicit description of the decomposition of a Task into sub-Tasks (for Decomposition Methods only)*
Execution	*Description of how the Method is performed (for Execution Methods only)*

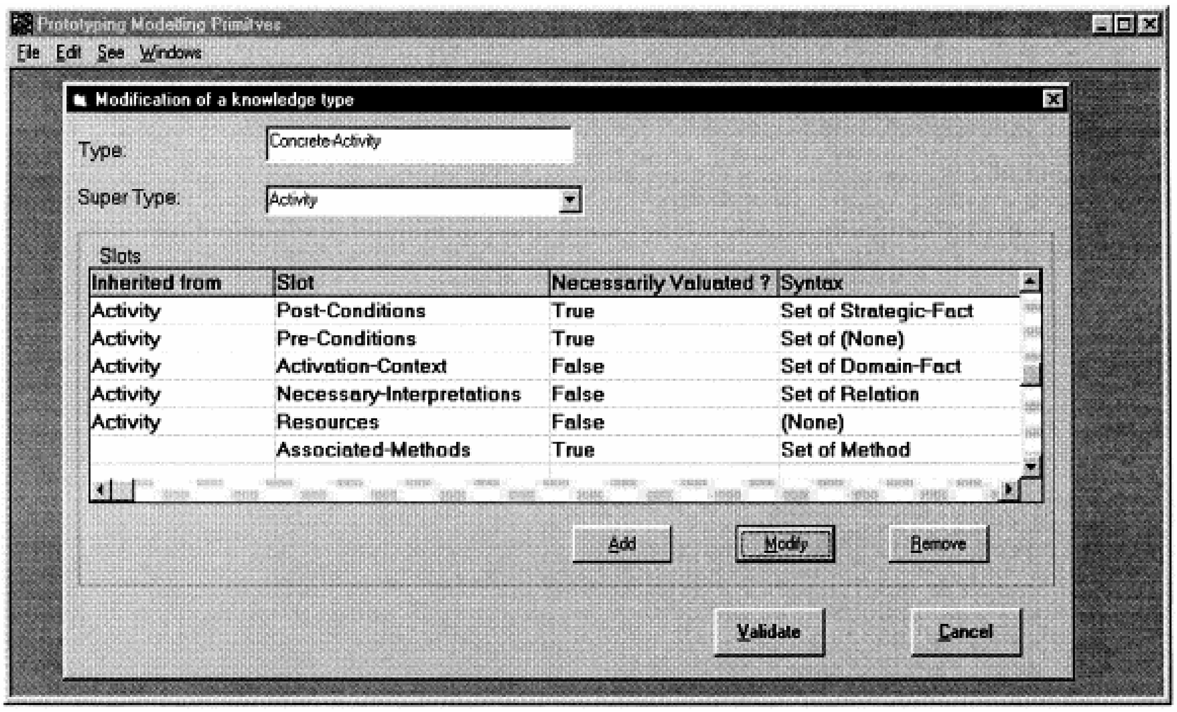
Figure 3. Description of a Method.

For Emma, the default definition of a Task is not satisfactory. First, as said before, the pertinence of a task is defined by taking into account aspects related to mathematical constraints and aspects related to the taught problem-solving method. Second, it is not *possible* to define what the expected results of a task are, but only an abstract characterization of what the state of the resolution is once the task is successfully achieved (*Post-Conditions*, in terms of strategic facts). Third, some interpretation knowledge must be attached to every task. Interpretation knowledge corresponds to a set of relations of the domain graph that should be inspected in order to interpret at an abstract level the results of a task, once it is achieved. For pedagogical reasons, one dissociates necessary interpretations (interpretations that must be considered when the task is achieved) and possible interpretations (interpretations that can be considered when the task is achieved). Finally, some tasks (in particular the highest-level tasks) are decomposed into subtasks but, at the moment this decomposition is achieved, one cannot yet define what these tasks effectively are. For instance, at the first step of a resolution, the task "Deal with an exercise" is decomposed into "Analyze the problem", "Formalize the problem", "Solve the problem" and "Examine the results". However, when this decomposition is achieved at runtime, one cannot yet know if "Formalize the problem" corresponds to the task "Formalize an optimization problem" or to the task "Formalize a statistic problem", as this depends on the results of the task "Analyze the problem".

In order to use operational structures that correspond to the paper-based modelling structures, the *default* definitions of the modelling primitives that are proposed by the kernel were adapted. The Task definition[4] was adapted by modifying some slots and adding some others: *Input-Context* is changed into *Activation-Context*, that denotes the pertinence from the point of view of mathematical constraints, and *Preconditions*, that denotes the pertinence from the point of view of the taught problem-solving method; two new slots, *Necessary-Interpretations* and Possible-*Interpretations*, are added. In order to respect the teacher's vocabulary, a task was renamed an Activity (we will use Task and Activity as synonyms). Two specializations of the Activity primitive were defined, the Prototype-Activity primitive (activities that are planned but cannot yet be made explicit, and therefore do not have associated methods) and the Concrete-Activity primitive (effective activities, that can be associated with methods; while resolution, when a prototype-activity is selected, the concrete-activity to be used to "instantiate" it is defined according to the context).

Concrete tasks and methods are defined by filling the description slots with a list of couples (fact, *value*) from the domain graph. The description of the Task and Method modelling primitives (and other primitives, if any) can be completed, explicitly, with an indication of what domain knowledge types can be used to fill the slots. For instance, in Emma, the *Post-Conditions* of an activity are supposed to be a set of *Strategic facts* (cf. Fig. 4). Checking if the knowledge-base modelling primitives instances respect their associated syntactical constraints helps in maintaining the description of the knowledge-base components at the same degree of abstraction.

[4] The DSTM default definition of a Method has also been modified in order to match Emma model. It should be noted that DSTM also allows the construction of additional modeling primitives if it appears necessary. For instance, in Emma, a Process primitive is used. It denotes concrete domain manipulations that are beneath the level that is to be made explicit.



Modification of a knowledge type

Type: Concrete-Activity

Super Type: Activity

Slots

Inherited from	Slot	Necessarily Valuated ?	Syntax
Activity	Post-Conditions	True	Set of Strategic-Fact
Activity	Pre-Conditions	True	Set of (None)
Activity	Activation-Context	False	Set of Domain-Fact
Activity	Necessary-Interpretations	False	Set of Relation
Activity	Resources	False	(None)
	Associated-Methods	True	Set of Method

Figure 4. The Concrete-Activity primitive.

Fig. 4 presents how a modelling primitive can be defined or adapted from an existing one and Fig. 5 presents an instance.

Name	Definition-of-the-list-of-equations
Post-Conditions	(list-of-equations . defined)
Pre-Conditions	(variables . defined) (constraints . defined)
Activation-Context	(all-constraints-linear . true)
Resources	(list-of-variables . valuated) (list-of-constraints . valuated)
Associated-Methods	Define-list-of-equations
Necessary-Interpretations	(are-equivalent (nb-equations=nb-variables . true) (square-system . true)) …
Possible-Interpretations	()

Figure 5. An instance of the Concrete-Activity primitive.

The abstract notions

The abstract notions can be viewed as a high-level implementation-independent vocabulary that bridges the gap between an abstract description of the system selection mechanisms in terms of high-level actions (the control) and the modelling primitives (task and method). From an implementation point of view, an abstract notion is reified by an operational code that denotes its semantics. The abstract notions proposed by DSTM kernel are:

- *applicable Task* (a task that can possibly be achieved);
- *achieved Task* (a task that was achieved by a method and whose objective is attained), *unsuccessfully considered Task* (a task for which all the possible methods were considered and none of them allowed it to be achieved) and *pending*

Task (a task for which none of the methods considered so far allow it to be achieved, but not yet considered methods still exist);

- *candidate Method* (a method that can achieve a task);
- *applicable Method* (a method that can be fired);
- *favorable Method* (a method that is particularly relevant).

The default definitions that were retained for the abstract notions are based on the default definitions that were retained for the modelling primitives. For instance, the definition adopted for the abstract notion *candidate Method* is: "a method M_i is a *candidate Method* for the achievement of a task T_i if and only if M_i has explicitly been defined as achieving T_i or if M_i produces the results that are expected for T_i". An abstract notion is defined by a logical connector (and, or, not) over a set of sub-operations. Sub-operations can be expressed as the application of predefined primitives (e.g. `check-domain-knowledge`[5], that allows testing values of facts in the domain graph during resolution, `belongs to` or `subset`) over the different slots of the modelling primitives. For instance, the definition "a task T_i is an *applicable Task* if its input context is satisfied" is represented by an operation whose body is "`check-domain-knowledge INPUT-CONTEXT task`". According to the modifications that were made at the modelling primitive level, the default definitions of the abstract notions can be adapted and/or some others can be created. In order not to complexify the definition of an abstract notion, one can first define support notions, i.e. intermediate notions, and then use these support notions to express the abstract notion. If necessary, new abstract notions can be defined in a similar way to the support notions.

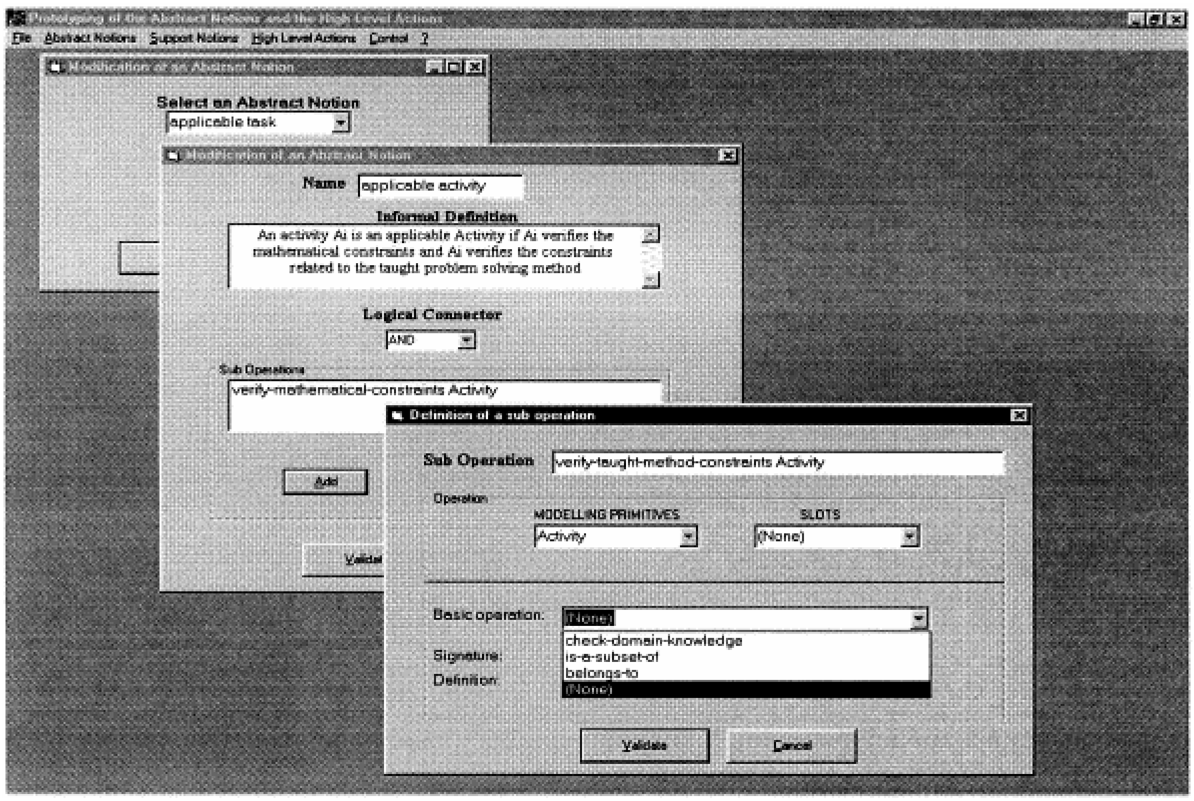
For Emma, the *applicable Task* abstract notion must be modified into the applicable activity abstract notion, whose definition is: "an activity A_i is an *applicable Activity* if it satisfies the mathematical constraints (modelled by the slots *Activation Context* and Resources) and the constraints related to the taught-method (modelled by the slot *Preconditions*). First, two support notions[6] were defined, *verify mathematical constraints activity* and verify *taught-method constraints activity*. Then, the *applicable Task* abstract notion was modified by changing its logical connector and its sub-operations. An *applicable Activity* is finally defined: `verify-mathematical-constraints Activity-i AND verify-taught-method-constraints Activity-i`. New abstract notions have also been introduced, such as *possible instantiation Activity*[7].

Fig. 6 presents how the default definition of an abstract notion can be modified (support notions and new abstract notions can be defined in a similar way). The interface hides the syntax of the operational language in which the different notions are represented, allowing their description at a level where one manipulates modelling notions (e.g. the slots of the modelling primitives or set primitives).

[5] A set of primitives such as `check-domain-knowledge` permits the connection of DSTM to the language used to model the domain. From DSTM point of view, using a particular domain representation language only requires the construction of the adapted connection primitives.

[6] The interest of defining a support notion can be to simplify the description of an abstract notion and/or to reify a pertinent aspect of the modeling (as abstract notions do). In the present case, these support notions were introduced because they correspond to modeling notions.

[7] The definition "a concrete-activity A_i is a possible instantiation Activity for a prototype-activity A_j if A_i has the same objective as A_j and A_j post-conditions are a subset of A_i post-conditions" is coded: `equal OBJECTIVE Activity-i OBJECTIVE Activity-j AND subset POST-CONDITIONS Activity-j POST-CONDITIONS Activity-i`.


Prototyping of the Abstract Notions and the High Level Actions
File
Abstract Notions
Support Notions
High Level Actions
Control
?
Modification of an Abstract Notion
Modification of an Abstract Notion
Definition of a sub operation
Add
Validate
Validate
Cancel

Select an Abstract Notion: applicable task

Name: applicable activity

Informal Definition: An activity Ai is an applicable Activity if Ai verifies the mathematical constraints and Ai verifies the constraints related to the taught problem solving method

Logical Connector: AND

Sub Operations: verify-mathematical-constraints Activity

Sub Operation: verify-taught-method-constraints Activity

Operation

MODELLING PRIMITIVES: Activity

SLOTS: (None)

Basic operation: (None)

check-domain-knowledge
is-a-subset-of
belongs-to
(None)

Signature:

Definition:

Figure 6. Modifying the applicable Task basic definition into applicable Activity.

The high-level actions

A high-level action corresponds to a selection mechanism based on a criteria that is denoted by an abstract notion. The high-level actions proposed by DSTM kernel are:

- *Select an applicable Task*: selects a task that can be achieved from a set of not yet achieved tasks. This action is based on the *applicable Task* abstract notion.
- *Identify candidate Methods*: identifies the methods that can achieve a task. This action is based on the *candidate Method* notion.
- *Identify applicable Methods*: identifies the methods that can be used from a set of methods. This action is based on the *applicable Method* notion.
- *Select a Method*: selects a method from a set of methods. This action is based on the *favorable Method* notion.
- *Evaluate the state of a Task*: evaluates the state of a task after the activation of a method. This action is based on the *achieved Task*, unsuccessfully considered *Task* and pending *Task* abstract notions.

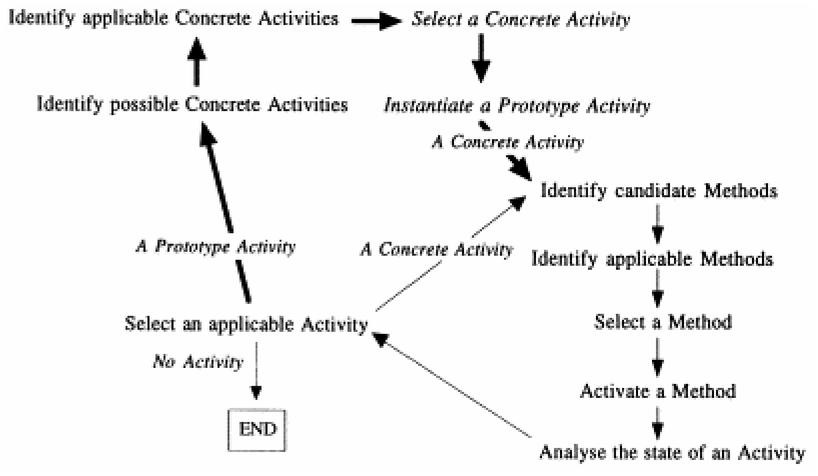
All the high-level actions have the same structure, their differences only stand in their signature and the abstract notions they manipulate[8]. According to what modifications were made on the abstract notions, detailed adaptations of some of the high-level actions and/or definition of new high-level actions are necessary. In Emma, the DSTM high-level actions that are reused (such as *Select an applicable Task*) only require modifying the abstract notion

[8] A high-level action is defined by constructions such as *match set1 set2 criteria*, where the primitive match constructs *set2* by selecting the items from *set1* that respect criteria and criteria is an abstract notion.

that is referred to (replacement of the default abstract notion *applicable Task* by the Emma abstract notion *applicable Activity*). This is a direct advantage of the limited-interaction principle. For new actions such as *Identify possible Concrete Activities*, the high-level actions directly proposed by the kernel can be used as patterns. These adaptations can be achieved through a similar interface to that of the abstract notions.

The control level

Different types of control (e.g. static decomposition of tasks or dynamic selection of tasks and methods) can be defined over these high-level actions. As an example, dynamic selection of tasks and methods, which is the default control[9] in DSTM, is defined as a simple iteration over the high-level actions listed previously. In the case of a static decomposition of tasks, the abstract notion *applicable Task* must be defined in order not to denote a matching of a task with the current problem-solving context, but a simple access in a stack. In Emma, the default control algorithm was modified in order to manage the new high-level actions (cf. Fig. 7).



Identify applicable Concrete Activities
Select a Concrete Activity
Identify possible Concrete Activities
Instantiate a Prototype Activity
A Concrete Activity
Identify candidate Methods
A Prototype Activity
A Concrete Activity
Identify applicable Methods
Select an applicable Activity
Select a Method
No Activity
Activate a Method
END
Analyse the state of an Activity

Figure 7. The control algorithm for Emma.

Summary

DSTM proposes different levels of flexibility: the modelling primitives, the abstract notions, the high-level actions and the control algorithm. Although some modifications imply others, the adopted limited-interaction principle allows limiting the propagation of modifications to what is necessary from a modelling point of view (there are no side-effects due to implementation features). A graphical interface facilitates these different modifications and allows the process to be managed at a model level (therefore, by knowledge-engineers). The overall construction of an operational KBS using DSTM can be summarized as presented in Fig. 8.

[9] An algorithm is a simple way to model a control over the high-level actions. Although it is sufficient for most of the cases, it nevertheless has a certain number of shortcomings. In particular, the same control is used for all the tasks and methods and during all the solving. As a consequence, aspects such as: "some tasks must be absolutely solved at once and when they are considered one should try all possible methods; some tasks should be reconsidered and put back into the list of not yet achieved tasks if the selected method fails" cannot be modeled explicitly. Another consequence is that a high-level action is always performed in the same way. It is not possible, for example, to adapt the way a method is selected according to the considered task and/or to a particular context. In order to remedy these shortcomings, the algorithmic control can be replaced by an explicit scheduler, modeled as a specific KBS. We have described such an approach in (Istenes et al., 1996).

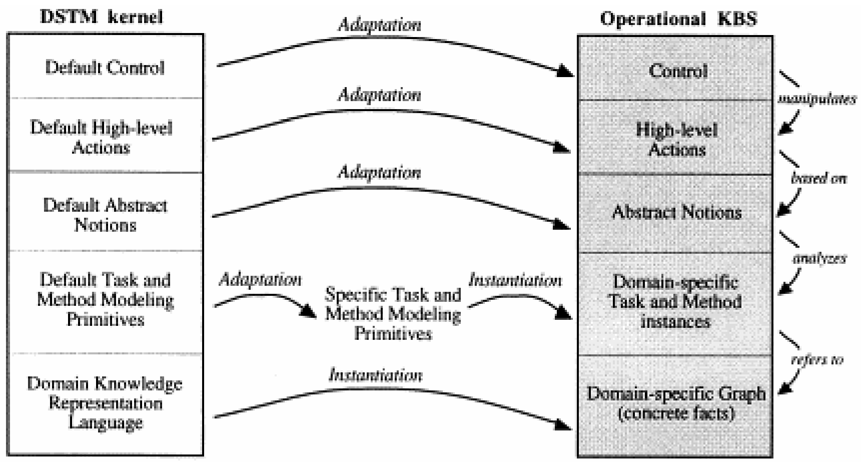

DSTM kernel
Default Control
Default High-level Actions
Default Abstract Notions
Default Task and Method Modeling Primitives
Domain Knowledge Representation Language
Adaptation
Adaptation
Adaptation
Adaptation
Specific Task and Method Modeling Primitives
Instantiation
Instantiation
Operational KBS
Control
High-level Actions
Abstract Notions
Domain-specific Task and Method instances
Domain-specific Graph (concrete facts)
manipulates
based on
analyzes
refers to

Figure 8. Constructing a KBS from DSTM kernel.

3.2. Refining the model with DSTM tools

The flexibility provided by DSTM enables the construction of an operational system that respects a structural correspondence with the paper-based model. A direct advantage is that the system problem-solving trace can be presented at the model level. The system can present what tasks and methods it selects and why, using the adopted modelling language. For instance, in Emma, the system can justify the fact it has not selected a particular activity by indicating that this activity is acceptable from a mathematical point of view, but not pertinent from the taught-method point of view. Proposing an explicit trace is a very ancient claim of KBS. We believe the flexibility provided by DSTM kernel, that allows constructing a system based on an adapted modelling language rather then on a “generic” modelling language, is more prone to allow the system to propose an understandable problem-solving trace than classical operationalization languages.

Although the fact that the system respects a structural correspondence with the model is a condition sine qua non to its use as a means to refine the model, it is not sufficient. The knowledge-engineers must be supported in their analysis of the model by tools that analyze the system from different points of views. In order to allow the construction of such tools, the different DSTM notions (modelling primitives, abstract notions, high-level action and control) are not translated into a low-level implementation language, but represented explicitly, using a dedicated reflective language. Therefore, the operational system can be analyzed by tools implemented as reflective modules and, as the system reifies the model, these tools can propose relevant information on the model.

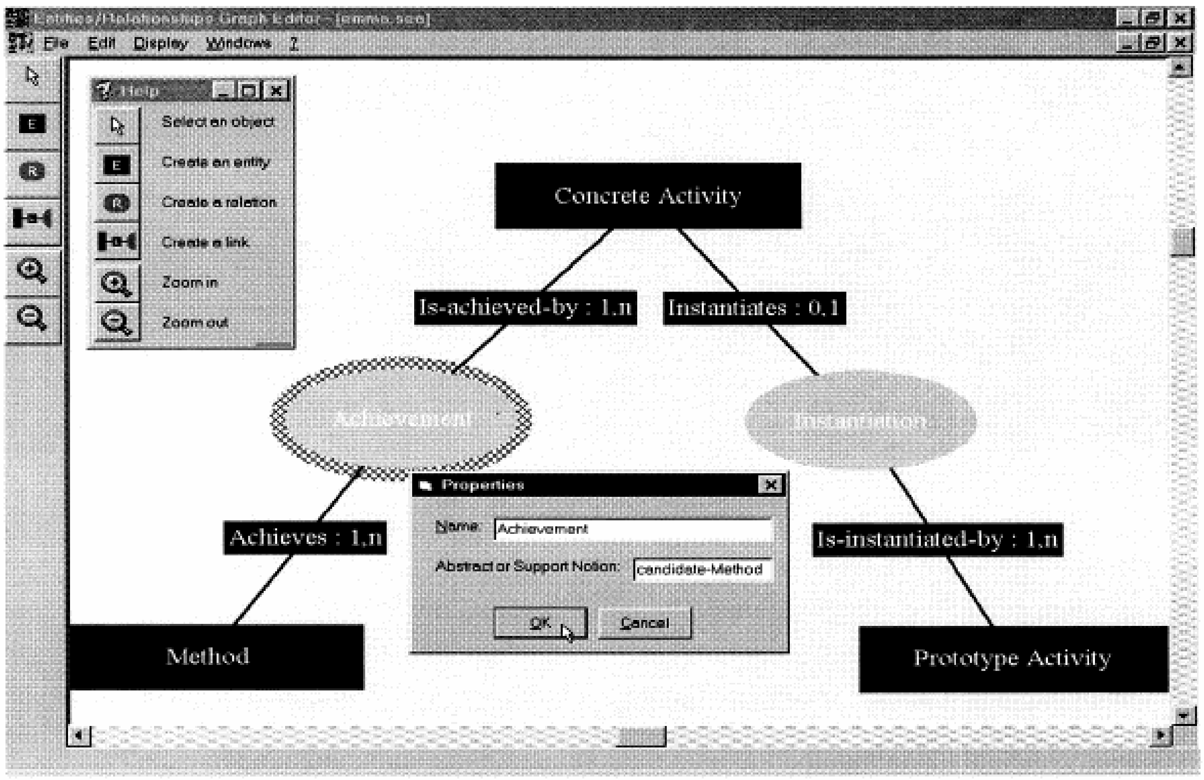
We present here below examples of such tools.

Checking constraints

Analyzing if the model respects some constraints is a verification activity: it aims at identifying “structural errors or errors of form in the system”, i.e. answering the question “am I building the product right?’ [Boehm, cited in Juristo, 1997].

Explicit constraints on the modelling primitives

DSTM framework allows the definition of integrity constraints that the modelling primitives’ instances must respect. Such constraints are defined by way of binary relationships between the modelling primitives. Defining these relationships consists in defining what role the notions of the model involved in the relationship play and the maximal and minimal cardinalities. For instance, in Emma model (cf. Fig. 9), the Instantiation relationship can be used to state that a Prototype-Activity must be instantiable by at least a Concrete-Activity (minimal cardinality on the *Is-instantiated-by* role) and that a Concrete-Activity can instantiate at most one Prototype-Activity (maximal cardinality on the *Instantiates* role). Once these constraints are defined, the Task–Method knowledge-base can be parsed in order to put into evidence violations of quantitative constraints such as "the *prototype-activity* A_i cannot be instantiated by any concrete-activity" or, using the Achievement relationship (cf. Fig. 9), "the *method* M_j is not defined as achieving any concrete-activity'.



Name: Achievement
Abstract or Support Notion: candidate-Method

Figure 9. Two integrity constraints on the Emma model.

The set of relationships that can be used to define the constraints to be checked is not predefined, but corresponds to the abstract and support notions that have been defined. From an implementation point of view, the operations that are used to check the constraints are automatically constructed, using the operations that operationalize the abstract or support notions as patterns. Therefore, this tool allows checking constraints related to the studied PSM model notions, i.e. to the adapted DSTM (it is not limited to the constraints that correspond to the defaults definitions).

Implicit constraints defined on the modelling primitives by the selection mechanisms

Some of the constraints that the Task and Method modelling primitives (and, then, their instances) must respect are due to the selection mechanisms that manipulate them. For certain mechanisms, the slots that are manipulated must be filled, an empty slot leading to an execution error. For some others, manipulating an empty slot can lead to an erratic behaviour of the system. For instance, an empty *Resources* slot can lead a method M_i to be unduly considered as firable.

In a “perfect world” context, the knowledge-engineers are well aware of these constraints. They can define them explicitly, when constructing the paper-based model, and (using DSTM editor) impose syntactical constraints on the modelling primitives (as shown previously). However, in a “real world” context, some of these constraints can be missed. Moreover, incoherence can appear if a modelling primitive (and, then, its instances) or an abstract or support notion (and, then, the selection mechanisms) is modified.

In order to check the implicit constraints that are defined by the selection mechanisms, we have developed analysis tools that can define what operations manipulate a given slot and how they manipulate it. Here again, the analysis is relative to the adopted modelling language (what definition was adopted for the modelling primitives and the selection mechanisms). For instance, let us suppose that, while refining the model, the knowledge-engineers decide to modify a support notion SN_i, that will now take into consideration the information denoted by the slot S_j (S_j being a slot originally defined for some other purpose). The reflective analysis of the selection mechanisms will calculate what new constraints exist and, for example, highlight that a particular method has no domain knowledge associated to its S_j slot, which can be problematic because the high-level action that performs the selection of methods uses the abstract notion applicable Method as a selection criteria, applicable Method uses the support notion SN_i, and SN_i checks if the domain knowledge associated to the S_j slot is verified.

It is interesting to notice that what is calculated here is the “role” (Reynaud et al., 1997) played by some particular knowledge on the way the problem-solving is processed[10]. This analysis essentially aims at detecting errors in the description of some of the tasks or methods of the, knowledge-base, typically problems raised by missing knowledge. Nevertheless, such a focus can also highlight problems related to how the mechanisms are defined. For instance, a problem detected for an instance that finally appears as correctly defined can highlight a problem in a selection mechanism (an abstract notion, a support notion). In such a case, what is highlighted is a modelling problem.

Analyzing the implemented problem-solving competence

An intrinsic difficulty of constructing a Task–Method model is that the problem-solving behaviour of the system is defined by the interactions of the different tasks and methods of the knowledge-base. Defining a new task (resp. method), modifying characteristics of some already existing tasks or modifying one of the abstract notions used in the selection mechanisms can influence the overall system behaviour. Keeping a synthetic understanding of how tasks and methods can interact can become overly complex. Information provided by a solving trace (i.e. what task is considered, why this task is an *applicable Task*, what method is used, why it is a *favorable Method*, etc.) allows analyzing a particular solving, but is not sufficient to enable an analysis of all the possible interactions that can appear, i.e. of the problem-solving competence that is effectively implemented.

[10] In an operational architecture dedicated to a particular PSM, the knowledge roles that are to be played in the PSM are known and the architecture can embed a knowledge acquisition tools based on these roles [Salt is a good example (Marcus and McDermott, 1989)]. In DSTM, how knowledge is manipulated can be changed and a tool that exploits how knowledge is used must, therefore, be based on an analysis of the manipulation mechanisms. A similar technique is used in Expect (Gil and Paris, 1996) to guide domain knowledge acquisition.

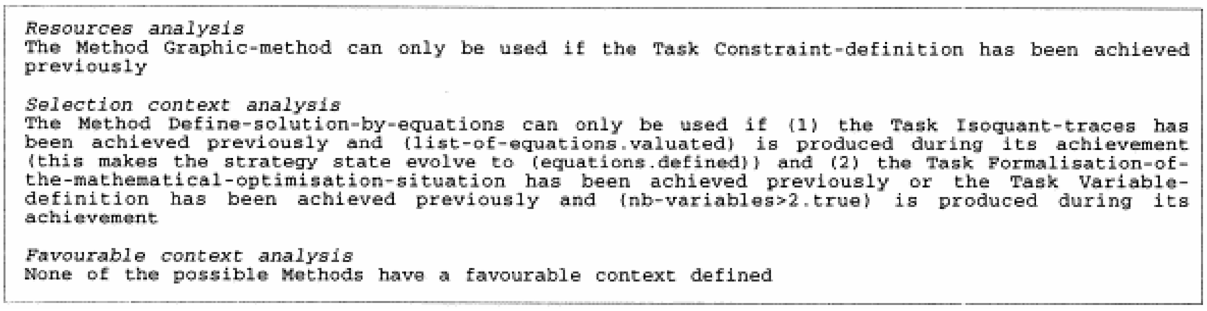
In order to help the knowledge-engineers to keep a synthetic understanding of the implemented problem-solving competence, they must be presented with information that corresponds to different axes of analysis, e.g. “Analyze how the achievement of the tasks preceding a task T_i influences the achievement of T_i“ or “Analyze how the achievement of the task T_i influences the achievement of the following tasks”. Analyzing the Task–Method knowledge-base from these different points of view helps studying if the model allows the expected problem-solving behaviour. It is a validation activity, that aims at answering the question “am I building the right product?” (Boehm, cited in Juristo, 1997).

Such analyses are in general too complex to be made “by hand”, as they do not just consist in picking-up information in the system (the considered task must be analyzed from the point of view of its interactions with other tasks, via the different possible methods). Therefore, we have constructed automated modules that analyze the Task–Method knowledge-base according to these analysis axes. We call these modules “explanation modules dedicated to the knowledge-engineers” (Trichet and Tchounikine, 1997b) to denote that what we want is to explain some consequences of the current description of the model, i.e. how the domain tasks and methods interact given the current selection mechanisms.

Here again, given the fact that the DSTM notions can be adapted, the construction of these explanations must be based on a reflective analysis of the implemented system (the modelling primitives, their instances and the selection mechanisms). Another difficulty is that how to achieve the analysis depends on some key-parameters that correspond to modelling decisions. For instance, when analyzing how a task can be achieved, one must take into consideration the fact that a task can be associated with multiple methods or with only one method, or that methods associated with a same task can have different influences on the rest of solving (i.e. side effects) or not.

In order to deal with this complexity, the construction of these explanations was modelled as a particular problem-solving task. General explanatory tasks (e.g. “Analyze how the achievement of the tasks preceding a task T_i influences the achievement of T_i“) have been decomposed into subtasks (e.g. “Analyze in what case the preceding tasks can conduct T_i not to be achieved” and “Analyzing the influence of preceding tasks on the selection of a method for T_i“). An explanatory-task is defined by its underlying explanation objective and the context in which it is relevant to be studied. This context is related to the different modelling actions that were performed by the knowledge-engineers within the DSTM framework. A modelling action can be concerned with the customization of the kernel (e.g. “modification of an abstract notion”, “modification of a modelling primative”) or the refinement of the current Task–Method knowledge-base (e.g. “definition of a new method”, “modification of an existing task”, etc.). An explanatory-task is associated with a set of explanatory-methods. An explanatory-method describes a way to reach an explanation objective according to the adopted modelling decisions. It can be a decomposition method (i.e. specify how a task is divided into sub-tasks according to the modelling decisions) or an operational method (i.e. identify the requested information and/or produce texts). This was modelled and implemented with DSTM.

Fig. 10 presents an episode of what is produced by the module “Analyze how the achievement of tasks preceding T_i influences the selection of a method for T_i“ in the context of one of Emma’s tasks. It should be noticed that the objective is not an automated correction: what is retrieved is not errors but information, its interpretation as an error is the knowledge-engineers’ responsibility. For instance, retracing a mistake identified because the context in which a particular method can be selected is considered as erroneous by the knowledge-engineers can lead to identifying a problem in the description of a particular domain task or method (a mistake in the filling of a slot) or a problem in the definition of an abstract notion (e.g. that the definition of what is an *applicable Method* is not satisfactory).



```
Resources analysis
The Method Graphic-method can only be used if the Task Constraint-definition has been achieved
previously

Selection context analysis
The Method Define-solution-by-equations can only be used if (1) the Task Isoquant-traces has
been achieved previously and (list-of-equations.valuated) is produced during its achievement
(this makes the strategy state evolve to (equations.defined)) and (2) the Task Formalisation-of-
the-mathematical-optimisation-situation has been achieved previously or the Task Variable-
definition has been achieved previously and (nb-variables>2.true) is produced during its
achievement

Favourable context analysis
None of the possible Methods have a favourable context defined
```

Figure 10. Information produced by an explanation module (episode).

The mechanisms that are used to operationalize the execution explanatory-methods reuse the ones that operationalize abstract notions and/or are constructed automatically from them. Therefore, from an implementation point of view, the explanation tool is not affected when the kernel is modified. Keeping the construction of the explanations coherent with the current model only requires the description of the modelling decisions that are used as criteria by the methods. A central advantage of considering the construction of explanations as a particular problem-solving task and to model and implement it with DSTM is the flexibility it allows. In fact, we use the general approach advocated in this paper (and DSTM capabilities) to incrementally refine our modelling of the construction of explanations, introducing in the description of the explanatory tasks and methods new dimensions that aim at modelling “explanation strategies”, i.e. strategies that define what explanation objective should be considered and how. The considered dimensions are the specificities of the model (general modelling decisions and lower-level considerations) and the interactions already achieved with the knowledge-engineers, using explanatory principles inspired from works related to explanations for the end-user (McCoy et al., 1991).

4. Discussion

4.1. Methodological background and scope of the approach

DSTM framework was constructed within the Mapcar project (Tchounikine, 1997a), whose general objective is to allow a constructive approach of expert knowledge modelling. When constructing a KBS, keeping close to human experts’ problem-solving is generally not necessary. Therefore, as modelling by abstraction is time consuming, most knowledge-acquisition methodologies [Kads (Wielinga et al., 1992), GT (Chandrasekaran and Johnson, 1993), Components of expertise (Steels, 1990), Protégé (Puerta et al., 1992)] advocate reusing (and eventually combining pieces of) predefined rational generic PSMs. However, some contexts require the construction of a KBS whose problem-solving behaviour is close to a particular human expert’ problem-solving behaviour, i.e. respects the relevant specificities of human expertise. This is the case when constructing a KBS used in an educational system (Emma is a good example). In such a case, the most rational model from a problem-solving point of view is not necessarily the most suitable: the KBS must solve problems as (or in a way that is coherent with how) teachers do when they interact with their students, that takes into account the specificities of how domain expertise should be taught, the pedagogical objectives and the pedagogical style (see Tchounikine, 1997b for further discussion). However, keeping close to how experts solve problems can also appear necessary to preserve the “know-how” of a firm, or to construct a conceptual model that remains meaningful for the experts from the firm and facilitates their interactions with the KBS (see Tchounikine et al., 1998 for a report on a realworld application in this context of “Corporate Knowledge”). In such cases, elaborating the conceptual model of the PSM by abstraction from domain experts’ knowledge helps in taking into account the idiosyncratic aspects of the expertise that should be modelled in the KBS. In order to help the experts (or teachers) to construct the PSM they would like to explicit and transmit, the Mapcar approach suggests a negotiation of the model between the experts (or teachers) and the knowledge-engineers. A prototype reifying the model serves as an unambiguous basis for this negotiation (a comprehensive presentation of the Mapcar approach can be found in Tchounikine, 1997a). DSTM flexibility and DSTM tools allow the putting into practice of this approach, that was used to construct Emma.

However, the interest of DSTM framework is not limited to the elaboration of a PSM conceptual model by abstraction from experts’ behaviour. As an operational framework, it can be used to operationalize any PSM modelled in terms of Tasks and Methods and, therefore, predefined generic PSMs [for instance, PSMs proposed by the CommonKads Library for expertise modelling (Breuker and Van de Velde, 1994)]. In such a context, DSTM will not be used to elaborate the model, but to refine it. Although reusing predefined PSMs is an appealing approach, a lot of practical problems appear when such models are applied to the real world, that require the refinement of the PSM (Speel and Aben, 1996). Moreover, as highlighted in (Cottam and Shadbolt, 1996), many of the generic PSMs that are proposed in libraries are system derived, i.e., model the rational problem-solving performed by existing systems. As a consequence, “their efficacy for human expert acquisition is debatable, and they may enforce an unsuitable system architecture upon the domain” (Cottam and Shadbolt, 1996). We believe DSTM flexibility can be used as a means to customize generic PSMs in order to match realworld constraints and/or better correspond to domain experts’ competence and, then, facilitate the acquisition of expert knowledge (its capacity to adapt the modelling language being particularly useful for such cases).

Finally, DSTM can be used as a kernel to generate specialized operationalization languages. As an example, it was used to create ZTM, an operationalization language dedicated to models constructed following the Macao methodology (Aussenac-Gilles and Matta, 1994). Here again, the flexibility of the framework was exploited to modify the modelling primitives and the selection mechanisms according the modelling choices adopted in Macao (Beaubeau et al., 1996).

4.2. Comparison with related works

Some of the methodologies that advocate reusing predefined PSM conciliate the modelling and the operationalizing phases by proposing predefined PSM skeletals and corresponding operational architectures. Such constructions can correspond to high-level well-known PSMs such as “heuristic classification” or “cover-and-differentiate” (the prototypical examples are the architectures constructed in the context of the RLM methodology, e.g. Salt (Marcus and McDermott, 1989) for the “propose-and-revise” PSM) or lower-level building blocks [GT (Chandrasekaran and Johnson, 1993), Protégé-II (Studer et al., 1996)]. With such built-in architectures, the operationalization phase is skipped or limited to a configuring problem. However, with such an approach, the PSM cannot be customized if some domain specificities (or context of use) require some aspects of the generic model to be adapted. Protégé-II aims at enabling some flexibility by proposing building blocks (called “mechanisms”) rather than a complete builtin architecture. However, the customization capacities remain limited by the proposed building-blocks. Our claim is not that predefined architectures or predefined set of “mechanisms” should not be used, but that such predefined constructions impose modelling constraints that can be to constraining. We see our approach as an alternative (or an additional level of flexibility to approaches such as Protégé-II) for such cases. Because they propose representation structures rather than PSMs' components, operationalization languages provide more flexibility. Some languages, such as Omos or Model-K, allow refining a Kads generic PSM while operationalizing it. Linster in particular has a similar claim to ours, and advocates "operationalization by continuous refinement" (Linster, 1993). Nevertheless, in Omos or Model-K, modifying the modelling primitives or how they are manipulated is not possible. In other words, the PSM model can be refined, but not the modelling language which is used to express it. As argued before, this unduly constrains the knowledge-engineers to use a predefined "generic" modelling language or leads to an implementation that does not respect a strict structural correspondence with the model (the model is compiled using syntactical tricks). Although modifying the modelling structures can appear somewhat time-consuming, it is in fact a good investment. When a satisfactory modelling language is defined, it greatly simplifies the acquisition of the domain expertise, because this expertise gently slips into slots that correspond to meaningful notions for both the knowledge-engineers and the domain experts. With DSTM, adapting the modelling structures only requires modifying identified parts of the kernel and can be done without low-level implementation problems, it is sufficiently easy to allow "prototyping" both the modelling language and the problem-solving model (we have presented the final Emma model, but several versions were tested)

4.3. Implementation

DSTM is implemented on top of Zola, a reflective language dedicated to the operationalization of conceptual models (Istenes and Tchounikine, 1996; Istenes, 1997). Modelling primitives are defined as Zola knowledge-types, concrete objects (e.g. domain tasks and methods) being defined as instances of these knowledge-types. The operations that reify the abstract notions, the support notions if any and the high-level actions are defined as Zola operations. For this purpose, Zola provides a set of primitives (e.g. `Match`, `Belongs-to` or `Subset`) that avoid spending time and getting confused with low-level implementation details. Details on the implementation can be found in (Trichet and Tchounikine, 1997a).

Zola reflective capacities are used to implement the analysis tools as meta-modules, i.e. modules that inspect the Zola implementation of the items to be analyzed. For instance, the analysis of the constraints implicitly defined by the selection mechanisms is achieved by analyzing the operations that operationalize the high-level actions and the abstract notions. The operations that check the integrity constraints defined with the graphical interface are dynamically created by other operations, that duplicate and modify the operations that operationalize the abstract notion underlying the relationship used to express the constraint. Compared with approaches that embed built-in tools, this "meta" approach has two central advantages. First, when the kernel is modified, the tools stay coherent. Second, new tools can be developed without modifying the kernel implementation.

At present, Zola is implemented in Lisp. An "industrial" version is under development using the Java language (Istenes et al., 1998), which will make available a Java version of DSTM.

4.4. Current direction of the work

DSTM kernel was tested in the context of different applications and we consider it as stabilized. In contrast, we do not consider the set of tools presently provided by DSTM framework as sufficient and we intend to extend it. Our "meta" approach permits developing new tools without modifying the rest of the system. This allows the evolution of the framework by constructing new “generic” tools such as the ones we have presented here, but also allows the construction of application-specific tools [examples of such tools constructed for Emma are presented in Choquet et al. (1997b), the interest of such tools was also demonstrated in Tchounikine et al. (1998)]. More generally, our objective is to make DSTM evolve towards an integrated framework. An ongoing work is the construction of an advisor system assisting the knowledge-engineers in their use of DSTM. The objective is to consider what actions were achieved (e.g. “Modification of the characteristics of the Task modelling primitive”, “Definition of a new abstract notion” or “Definition of a new method in the knowledge base”) and to advise a particular action to be achieved or the use of a specific tool. First, the advisor system confronts the actions that were achieved with a set of rules that models how-to-use DSTM principles, e.g. “if an abstract notion A_i is removed and A_i is used as a selection criteria in a high-level action H_j, then advise the modification of the high-level action H_j“. This first step aims at listing the different actions that have to be done to obtain a coherent operational system. Second, the advisor system confronts the modelling actions with the available DSTM tools. As an example, checking the cardinalities applied to the roles of a relationship R_i will be recommended when the abstract notion underlying the roles of R_i and/or the entities of R_i were modified.

5. Conclusions

The work we have presented in this paper is based on the assumption (correlated by personal and reported in the literature feedback of real world experiences) that the general structure of a PSM can (and must) be defined “on paper”, but that new problems (or new points of view on theoretically already tackled problems) appear when instantiating the model and implementing the system. Therefore, in addition to the flexibility features and the verification and validation facilities that can be used on paper-based models, the operationalization phase must allow and facilitate tackling these problems at the model level (and not by implementation patches). This requires two different types of functionalities. First, it is necessary to allow the knowledge-engineers to use the modelling language they think is the best adapted, and to use an operationalization language that corresponds to the adopted modelling language. Second, it is necessary to help the knowledge-engineers to master the complexity of the knowledge-base (in a Task–Method model, how tasks and methods can interact). We believe that constructing operationalization frameworks based on such ideas will help bridging the gap that exists between satisfactory paper-based models and satisfactory operational KBSs. The construction of the DSTM framework is an attempt to allow such an approach. It is based on a kernel that offers predefined but adaptable modelling and operationalizing structures and verification and validation tools defined as metatools.

6. References

Aussenac-Gilles, N. and Matta, N., 1994. Making the method of problem solving explicit with Macao. *International Journal on Human–Computer Studies* 40, pp. 193–219.

Beaubeau, D., Aussenac-Gilles, N. & Tchounikine, P. (1996). Mona au pays des rôles: opérationalisation de modèles conceptuels Mona en Zola. Research Report IRIT/962311, Institut de Recherche en Informatique de Toulouse (in French).

Breuker, J., & Van de Velde, W. (1994). CommonKADS library for expertise modelling. IOS Press, Ohmsha.

Chandrasekaran, B., & Johnson, T. (1993). Generic tasks and task structures: history, critique and new directions. In J. David, J. Krivine, & R. Simmons (Eds.), Second generation expert systems (pp. 232–272). Berlin: Springer.

Choquet, C., Tchounikine, P., & Trichet, F. (1997a). La modé1isation de la méthode de résolution de problème dans le système Emma. In Actes des Journées Francophones EIAO de Cachan (pp. 263–275). Paris: Hermès.

Choquet, C., Tchounikine, P., & Trichet, F. (1997b). Training strategies and knowledge acquisition: using the same reflective tools for different purposes. In 8th International Portuguese Conference on Artificial Intelligence (EPIA'97), number 1323 in Lectures Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science) (pp. 119–129). Coimbra, Portugal: Springer.

Cottam, H., & Shadbolt, N. (1996). Knowledge acquisition for search and rescue. In Knowledge Acquisition for Knowledge-Based Systems Workshop (Banff'96). Banff, Canada.

David, J., Krivine, J., & Simmons, R. (1993). Second generation expert systems. New York: Springer.

Eriksson, H. and Musen, M., 1993. Metatools for knowledge acquisition. *IEEE Software* 10 3, pp. 23–29.

Fensel, D. (1995). The knowledge acquisition and representation language KARL. Boston: Kluwer.

Gil, Y., & Paris, C. (1996). Towards method-independent knowledge acquisition. Knowledge Acquisition, 6(2).

Greboval, C., & Kassel, G. (1992). An approach to operationalize conceptual models: the shell AIDE. In European Knowledge Acquisition Workshop: Current Developments in Knowledge Acquisition (EKAW'92) (pp. 37–54). Berlin: Springer.

Greboval, C., & Kassel, G. (1994). The production of explanations seen as a design task: a case study. In European Conference on Artificial Intelligence (ECAI94) (pp. 351–355).

Guerrero-Rojo, V. (1995). MML, a modelling language with dynamic selection of methods. In Knowledge Acquisition for Knowledge-Based Systems Workshop (Banff'95). Banff, Canada.

Istenes, I. (1997). ZoLa: un langage réflexif pour représenter et opérationaliser des modèles conceptuels. Thèse de doctorat, Institut de Recherche en Informatique de Nantes, Université de Nantes (in French).

Istenes, I. and Tchounikine, P., 1996. Zola: a language to operationalise conceptual models of reasoning. *Journal of Computing and Information* 2 1, pp. 689–706.

Istenes, I., Tchounikine, P., & Trichet, F. (1996). Dynamic selection of tasks and methods m a knowledge level reflective activity. In A. Ramsay (Ed.), Artificial Intelligence: Methodology, Systems, Applications (AIMSA'96), number 35 in Frontiers in Artificial Intelligence and Applications (pp. 168–177). Sozopol, Bulgaria: IOS Press.

Istenes, Z., Trichet, F., Camilleri, G., & Fortes, A. (1998). Modelling knowledge structures and patterns: capturing world subsets through language structured subsets. Research Report IRIT/98-0611, Institut de Recherche en Informatique de Toulouse.

Jacob-Delouis, I. and Krivine, J., 1995. LISA: un langage réflexif pour opérationaliser les modèles d'expertise. *Intelligence Artificielle (In French)* 9(1), pp. 53–88.

Juristo, N. (1997). A common framework for conventional and knowledge based software validation and verification. In 9th International Conference on Software Engineering and Knowledge Engineering (SEKE97) (pp. 287–294). Madrid, Spain.

Karbach, W., & Voss, A. (1993). Model-K for prototyping and strategic reasoning at the knowledge level. In J. David, J. Krivine, & R. Simmons (Eds.), Second generation expert systems (pp. 721–745). Berlin: Springer.

Linster, M., 1993. Closing the gap between modelling to make sense and modelling to implement systems. *International Journal of Intelligent Systems* 8, pp. 209–230.

Marcus, S. and McDermott, J., 1989. SALT: a knowledge acquisition language for propose-and-revise systems. *Artificial Intelligence* 39(1), p. 137.

McCoy, K., Moore, J., Suthers, D., & Swartout, B. (1991). AAA1-91 Workshop on Comparative Analysis of Explanantion Planning Architectures.

Newel, A., 1982. The knowledge level. *Artificial Intelligence* 18, pp. 87–127.

Puerta, A., Egar, J, Tu, S. and Musen, M., 1992. A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition* 4, pp. 171–196.

Reinders, M., & Bredeweg, B. (1992). Strategic reasoning as a reflective task. In International Workshop on Reflection and MetaLevel Architecture. Tamacity, Tokyo.

Reinders, M., Vinkhuyzen, E., Voss, A., Akkermans, H., Balder, J., BartschSporl, B., Bredeweg, B., Drouven, U., van Harmelen, F., Karbach, W., Karsen, Z., Schreiber, G. and Wielinga, B., 1991. A conceptual modelling framework for knowledge-level reflection. *AI Communications* 4 23, pp. 74–87.

Reynaud, C., Aussenac-Gilles, N., Tchounikine, P., & Trichet, F. (1997). The notion of role in conceptual modeling. In R. Benjamins and E. Plaza (Eds.), 10th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW'97), number 1319 in Lectures Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science) (pp. 221–236). Sant Feliu de Guixols, Spain: Springer.

Speel, P., & Aben, M. (1996). Applying a Library of Problem Solving Methods on a RealLife Task. In Knowledge Acquisition for Knowledge-Based Systems Workshop (Banff'96). Banff, Canada.

Steels, L., 1990. *Components of expertise. AI Magazine* 11 2, pp. 29–49.

Studer, R., Eriksson, H., Gennari, J., Tu, S., Fensel, D., & Musen, M. (1996). Ontologies and the configuration of problem-solving methods. In Workshop on Knowledge Acquisition for Knowledge Based Systems (Banff'96). Banff, Canada.

Talon, X., & Pierret-Golbreich, C. (1996). TASK: a framework for the different steps of a KBS construction. In Workshop on Knowledge Engineering and Modelling Language (KEML'96). Paris, France.

Tchounikine, P., 1997. Mapcar: a framework to support the elaboration of the conceptual model of a knowledge based system. *International Journal of Intelligent Systems* 12 6, pp. 441–468.

Tchounikine, P., 1997. Modelling problem-solving for an educational system. *Intelligent Tutoring Media* 7 34, pp. 83–96.

Tchounikine, P., Choquet, C., & Istenes, Z. (1998). Elaborating the problem-solving model of a fault diagnosis expert system by knowledge level prototyping. *Expert Systems with Applications*, 14(23).

Trichet, F. (1997). Trois outils d’aide à la construction d’un Système à Base de Connaissances en Zola. Research Report IRIN163, Institut de Recherche en Informatique de Nantes (in French).

Trichet, F., & Tchounikine, P. (1997a). Reusing a flexible task–method framework to prototype a knowledge based system. In 9th International Conference on Software Engineering and Knowledge Engineering (SEKE’97) (pp. 192–199). Madrid, Spain.

Trichet, F., & Tchounikine, P. (1997b). Structured explanations as a support to model problem-solving in a Task–Method paradigm. In G. Grahne (Ed.), Sixth Scandinavian Conference on Artificial Intelligence (SCAI’97), number 40 in Frontiers in Artificial Intelligence and Applications (pp. 131–142). Amsterdam, Holland: IOS Press.

van Heijst, G., Schreiber, A. and Wielinga, B., 1997. Using explicit ontologies in KBS development. *International Journal of Human Computer Studies* 42 23, pp. 183–292.

Vanwelkenhuysen, J., & Rademakers, P. (1990). Mapping a Knowledge Level analysis onto a computational framework. In European Conference on Artificial Intelligence (ECAI’90) (pp. 661–666).

Wielinga, B., Schreiber, A. and Breuker, A., 1992. KADS: a modelling approach to knowledge engineering. *Knowledge Acquisition* 4, pp. 92–116.