Knowledge level reflection

Aurelien Slodzian

VUB AI LAB
e-mail: as@arti.vub.ac.be

May 17, 1994

Abstract

This document presents a theory for knowledge level reflection together with experimental results, proving that it is possible to design knowledge systems intended to build, verify or control other knowledge systems. Furthermore, this may be done in a very simple way and still produce surprisingly efficient results.
Keywords: Knowledge level reflection, componential methodology, meta-level, reuse.

This project was carried out under the direction of **Prof. Dr. Luc Steels** and **Dr. Walter Van de Velde**.

Master thesis
Faculty of Sciences
Master in Artificial Intelligence
Vrije Universiteit Brussel
Academic year 1993 – 1994

Contents

1 Introduction

An important aim in knowledge engineering is to make the computer participate more in the development of knowledge systems. In this document, we report on a concrete experiment in using the computer as a knowledge engineering assistant (figure 1). This was carried out using KresT, a knowledge-system design tool implementing the componential methodology [Steels, 1990]. According to this methodology, the basic components of a knowledge-level description are tasks, models, methods and their relations. This tool also provides effectively a continuous (but manual) design control, from knowledge level modelling [Newell, 1990], [Van de Velde, 1993] to application execution.

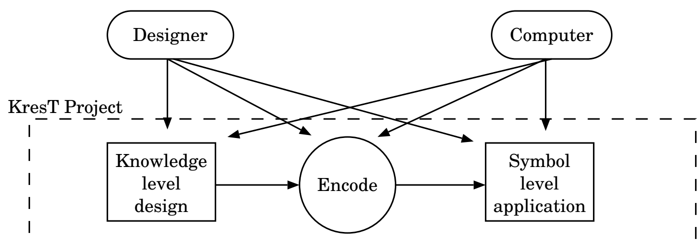

Figure 1: Computer aided modelling

The efficiency of the computer's participation in design depends upon its relevance. Many knowledge design environments provide specific hard-coded utilities such as validation, verification, coding and so on. Although they may support some concrete knowledge engineering activities, engineers typically apply a personal variant of some methodology. Moreover, there will be many situation specific variations. Therefore, for the computer to provide meaningful support it should rely upon a description of the particular engineer's goals in the context of his ongoing project.

In other words, the engineer needs two different tools: one to build the knowledge system, and one to teach the building strategy. For simplicity reasons, it is furthermore expected that these tools are provided by one and the same environment, and that they use the same language (as figure 2 reports, in the context of KresT).

The concrete objectives of this project are firstly to propose such a design environment and secondly to experiment it. This environment should conform to the following requirements:

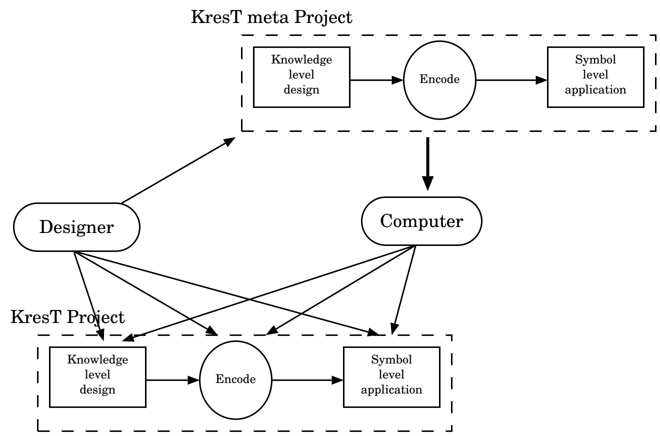

Figure 2: Design and meta-design

1. the engineer will design knowledge systems in a knowledge-level way;
2. he will also describe parts of the design process, also in a knowledge-level way;
3. the computer will use those descriptions as a guide to effectively participate in the design.

The description language used by KresT already allows one to identify and manipulate fragments of knowledge level models. This paper presents an extension of KresT, the MetaKit, that additionally permits to describe one's design strategy and makes the system effectively use those descriptions in subsequent developments. More specifically, it is possible to describe in full detail, but still readably, the following tasks of the knowledge engineer:

1. Construct a knowledge level description of the knowledge system, either from scratch or from reusable components.
2. Verify this description (check for inconsistencies, or incompleteness)
3. Implement the system, acquire the expert's knowledge.
4. Test the implementation (w.r.t. behavioural requirements).
5. Change the knowledge-level design according to the results of those tests.

This already covers a broad range of knowledge engineering activities but is in no way a restriction on the MetaKit's possibilities.

Section 2 presents a preliminary clarification of theoretical issues in the componential methodology and introduces the concepts of meta-level and knowledge-level reflection in the context of our project. The main point here is not to study knowledge-level reflection in itself, but to use it to prepare a consistent framework for modelling the design process. Then, in Section 3, the basics of KresT are explained and the MetaKit is introduced.

The following two sections describe application examples, covering topics like verification of knowledge-level designs (Section 4) and definition of modelling strategies (Section 5). Finally, the approach is applied to the specific problems of the symbol-level: modelling the implementation phase of knowledge-level descriptions and analysing, from the knowledge-level, the so generated programs (Section 6).

Unless otherwise stated, all examples given in this document were effectively implemented and tested; sometimes in a slightly different way than reported, for the sake of clarity. They were all inspired from practical problems encountered in knowledge system design.

2 The meta-level and the knowledge-level

As stated in the introduction, we expect a computer to perform a part of the knowledge engineer's work. This requires some analysis of this work, mainly to avoid problems which could arise from the fact that one machine will be, at the same time, the "pencil" and the "hand". Moreover, if some intrinsic limitations exist, they must be known in advance.

The analysis will be conducted considering the knowledge engineer as an expert who produces *knowledge-level* and *symbol-level* descriptions of an agent. Because his work is to describe, the knowledge engineer will be said to operate at a *meta-level*. This level is not another level of description but a level of operation. How these levels are related is the main topic of this section.

The answer will provide us with a compact kernel of clearly related concepts, allowing to analyse the way a computer performs the tasks it is entrusted with. The results of this analysis were effectively used as a guide in designing the MetaKit itself, as well as for validating the experimental results.

The description levels are further refined by the componential methodology in section 2.1. Then the concept of meta-level is introduced, related to the other levels and placed on an operational ground (section 2.2).

2.1 Description levels

The concepts of knowledge and symbol levels were introduced long ago [Newell, 1982] and since then have been broadly used, albeit with many different interpretations. In the current context, a *knowledge-level description* is a structured description of an agent's goals, knowledge and means to use this knowledge to achieve those goals. Such a description should not be concerned with the representation of the agent's knowledge nor with the concrete mechanism used to perform the tasks. In contrast, a *symbol-level description* (or *program*) is an explicit mechanisation of those tasks, with complete indications on representation means [Newell, 1990], [Van de Velde, 1993].

These concepts are further refined by the componential methodology [Steels, 1990], which provides us not only with an ontology of knowledge level design, but also with guidelines to extend this ontology to the concepts related to knowledge level reflection, as we will see below.

According to this theory, we call *project* a description of an agents knowledge and goals; this analysis of the agent being built of *components* of three types:

- *tasks* describe the agent's goals and subgoals;
- *methods* describe how the agent may reach his goals;
- *models* describe the knowledge used or produced by the agent while performing his tasks.

Those components may be linked together by the following relations:

- tasks may form a task-subtask hierarchy
- a method has to be associated to each task
- models play certain roles with respect to methods (*method-role* relation)
- tasks also play roles with respect to methods; in particular to task decomposition methods.
- models may also be connected to tasks independently of the methods; this indicates the necessary use of the model to perform the task, whatever the method is.

The descriptions of instances of those relations are considered as parts of the descriptions of the involved components. Thus, when a model is connected to a task, the model's and the task's descriptions have to be updated accordingly. All relations, excepted method-roles, exist only through their instances. On the contrary, the *method-role* relations owe their existence to the algorithm of the methods. They are thus part of the method's description before the link with a task or model is established. The method is said to be the *owner* of the role while the model or the task is its *filler*. One should avoid confusion between the *subtask* relation (between tasks) and the *subtask* method-role (between methods and tasks). *Task roles* are defined in the same way.

In addition to these knowledge-level concepts, we will manipulate *programs*. They are the symbol-level descriptions of an agent, in some language emphasising the mechanistic aspects. A program may thus be compiled and executed on a computer. The part of the computer directly involved in the execution of a program is said to be the *execution level* description of the agent. This level is considered as a sub-level of the symbol level.

the code-level is a textual specification of the mechanisms and information involved in the agent. It is descriptive and, theoretically, machine independent. It is static by nature.

the execution-level is an internal computer representation of the same thing. Such a representation is generally the result of loading a code-level description into a given computer and, if possible, of running it. It is thus subject to modification due to interactions with the system's users.

The code and execution-levels are in fact two different symbol-level languages, considered as sub-levels because they may exist at the same time, in the same environment. It is common practice to identify the symbol-level and the code-level.

These levels appear in KresT's encoding procedure (figure 3): knowledge-level descriptions are used as specifications to automatically build code-level (Lisp) programs which may in turn be loaded into the computer's memory to become execution-level objects. Once the program is loaded, knowledge acquisition may occur and the effective value of the representation of the agent's knowledge will change. KresT allows to transfer at the code-level the modifications which occured at the execution-level. The knowledge-level description is not affected this procedure because it is only concerned with the structure of the symbol-level representation and not the actual content of the knowledge.

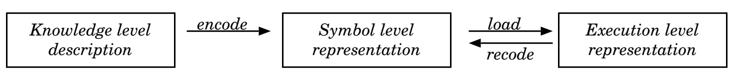

Figure 3: Flow of encoding procedure

2.2 The meta-level

The level at which the knowledge engineer works when he builds a knowledge-level description of an agent is called the *meta-level*. The counterpart of this concept is the *object-level*, at which the agent's behaviour takes place. Such a distinction is possible because the engineer is obviously not operating in the agent's world but as if he was on a higher observation point.

The concept of meta-level has been studied for a long time, since the discovery of inconsistencies in Set Theory at the beginning of the century [Gödel, 1931], [Tarski, 1972], [Putnam, 1989a], [Putnam, 1989b], [Ladriere, 1957]. This lead to a number of theories in computer science, some with successful results in computational reflection [Maes, 1987], [Ferber, 1988], [Carle, 1992]. This number of theories calls for a clarification of the current approach. Moreover, in the case of knowledge-level design, things get much more complex and the slightest theoretical error leads to dramatic inconsistencies in applications.

Let's consider only the designer and the agent he is describing. This description is first made at the knowledge-level, then encoded into a code-level program which is in turn loaded and executed. At this point, the designer made the computer behave the same way as the originally described agent, with a degree of precision depending on the design's grain size and completeness. So, the execution-level of the original project and the agent's operations are both behaviours at the same object level. The former is a simulation and thus a representation, of the latter. For similar reasons, the code-level may also be considered as a representation of the agent; the difference with the execution-level lying in the temporal aspects of these sub-levels: the code-level is static by nature while the execution-level is dynamic (figure 4). Note that there is no possible confusion in considering the symbol sub-levels as representations: the action of representing is a projection, in this case from the agent's world (Euclidian space) into the computer space (symbol space). On the other hand, knowledge-level descriptions are intended to describe the agent, not to represent it. They are thus at a meta-level with respect to the agent's behaviour. In the case where knowledge-level descriptions are encoded into programs, they will also describe the structure of the symbol-level representations which derive from them.

If the designer then wants to analyse his own design, in our context with the purpose to describe it, then the properties investigated will be the ones of the design itself and he has to change from agent related vocabulary to pure design vocabulary. In that case the designer works at a level above the normal design work, and thus shifts one meta-level up. In other words, he becomes the observed agent.

In more general cases, when the complexity of the engineer / agent relation increases, the same type of inter-level relations holds:

> *Analysis of the agent's behaviour takes place at a meta-level above the agent but at the execution-level of the observer.*

This sort of loop-back is the foundation of our reflective constructions:

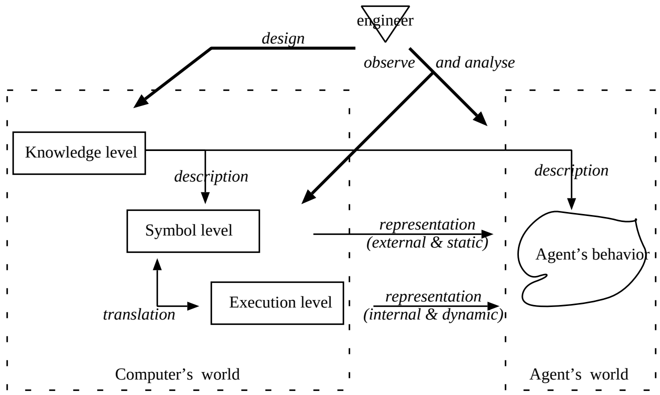

Figure 4: Relative position of levels

as any agent performing an action, the design process may be viewed from above and gives rise to a meta-level. From that level, one may describe the design process using a knowledge-level methodology. The execution-level (if any) of that description will always be at the same level as the described process.

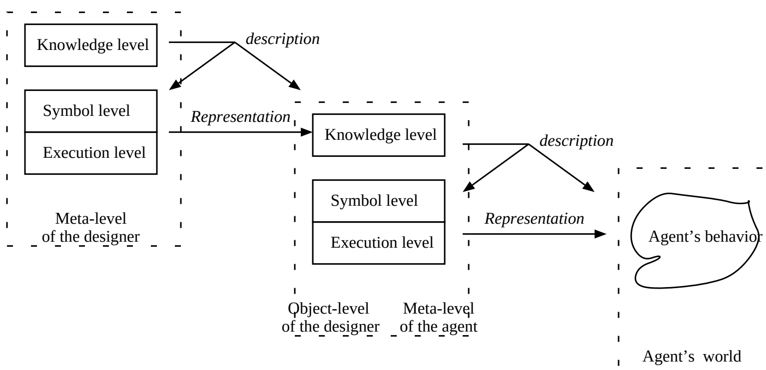

Figure 5: Basic scheme for the meta-level

Describing the design process is done from the design's meta-level. In a similar way, we can say that controlling the execution of a program is done from the execution-level's meta-level. More generally, every activity level gives rise to a meta-level. This is free of inconsistencies as long as we keep in mind the following working restriction.

A meta-level may exist only above a level at which some behaviour appears.

This rule eliminates the possibility of considering a meta-level directly above a meta-level, which is a common cause for interpretation problems. On the other hand, operations like knowledge-level modelling, which already take place at a meta-level (w.r.t. the agent), do still have a meta-level because they may be considered as behaviours.

In a knowledge-level modelling perspective, the only meta-level operations that will be considered are the ones which are somehow related to a knowledge level description. These operations fall into the three following categories.

1. build a knowledge-level description of the object-level
2. derive from it a code-level representation of the object-level
3. run the execution-level representation.

For example, directly representing the agent's behaviour in a computer system is a meta-level operation in the general sense, but we restrict our study to the cases where such an implementation follows a knowledge-level description. This restrictive definition of the meta-level has the main advantage to provide us with minimal conditions to build knowledge-level descriptions of the design process. According to this scheme, we can make a primary task decomposition of the designer's work, as shown in figure 6 and table 1, in which we consider that the agent performing the tasks is the designer. This also shows how sources of inconsistencies are removed: being the observed agent the expert himself does not appear in that scheme, only a knowledge-level description of expert's work may be taken into account.

This task decomposition has of course to be further refined by each knowledge engineer, according to his precise needs. It is precisely the goal of the next sections to present generic descriptions of some of those subtasks. In terms of KresT, this task decomposition forms the skeleton of a project, namely a *meta-project*. These tasks may be grouped in three categories, according to the three types of meta-level operations listed above. Obviously the **build** and **verify** tasks are of type 1 while **encode** is of type 2 and **run** and **validate** are of type 3. This classification propagates to their respective subtasks. It may furthermore be expressed in terms of levels instead of class types. For example, the **build** task and its subtasks describe in fact the

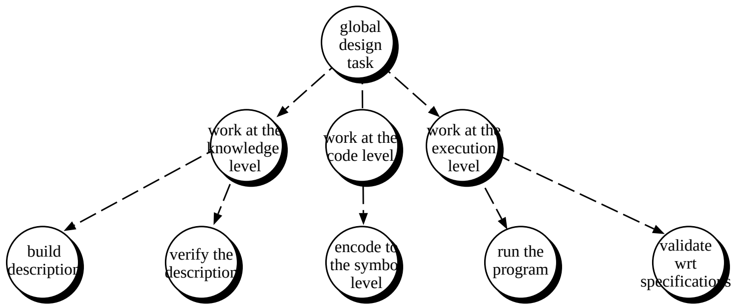

Figure 6: The main meta-tasks

task	goal
build	construct a knowledge-level description of the agent
verify	check the description against inconsistencies
encode	derive the code level representation
run	run, debug the program
validate	compare program result to expectations

Table 1: Summary of the main meta-tasks

goals of the knowledge engineer. Designing those tasks is thus at a meta-level above the knowledge-level phase of the design process. Similarly, the **encode** is at a meta-level above the conversion to code-level phase. This distinction between the meta-level activities may be expressed in terms of sub-levels of the meta-level.

1. the *meta-knowledge-level* is the one at which the knowledge-level modelling activities are described (tasks of type 1).
2. at the *meta-code-level* , the code-level representations are produced, analysed or modified (tasks of type 2).
3. the *meta-execution-level* gathers the meta-operations directly concerned with the behaviour of the agent. It isthus a level at which the execution of the program is focused on, and, consequently, the level at which tasks of type 3 are described.

What has been presented so far, is the concept of meta-level, its relations with usual description levels of knowledge-based systems and its first consequences on the design of meta-projects. We will say that knowledge level reflection occurs when a single system (for example the designer) analyses his own productions, that is when the object-system is identified with its meta-level model. In particular, this includes the case where the knowledge engineer makes, in the same environment a description of a system and its meta-level.

2.3 Computational reflection

The difference between this approach and computational reflection lies in the multiplicity of the description levels: according to the previous definitions, one may say that in computational reflection only the symbol-levels of the object-level meta-level do exist. The reflective part of such systems consists in procedures that manipulate the current state of the running programs. As noticed in [Wielinga and van Harmelen, 1993], this requires that the object and meta levels are causally connected. On the contrary, the proposed scheme involves one or more systems making models and representations of some other ones, with no need for causal nor physical connection.

Moreover, the specificity of computational reflection is that the meta-level is a part of the architecture of the computer system and concerns the computer's behaviour [Maes, 1987]. In our case, the computer is a medium for descriptions and representations; it is in no case the object of any analysis. See also [Laird et al., 1987], [Smith, 1984]

2.4 Conclusion

In [Wielinga and van Harmelen, 1993], some relevant issues are raised which help to characterise reflective knowledge systems; as a summary of our approach, these questions are answered here.

Nature of the object-model (ie which representation the meta-system has of the object-system)

The meta-system may have knowledge-level, code-level or execution-level views of the object system. There are no further statements about the description and representation languages used for that. All three levels of description (knowledge, code and execution) are completely independent in nature of the object system.

Separation vs. amalgamation (are the meta- and object-systems separate or is one a sub-part of the other ?)

Both systems are clearly separated. The meta-system makes descriptions of an object system which is considered to be independent of it. This does not prevent both systems to be present in one and the same environment.

Is there a causal connection ? (ie what are the respective influences of each system on the other ?)

There is no more causal connection here than between the designer and his subject. Moreover, descriptions of design tasks do no necessarily depend on what is designed (for example, verification of designs is domain independent). This proves the independence of both systems.

Where is the locus of action ? (are the meta and object systems working simultaneously, or each in turn ?)

This question is not really relevant here because the absence of causal connection implies that there are no temporal relations. Of course, the analysis of the agent's behaviour must occur during or after his work. And the design of a knowledge-level model of this system must obviously take place before the execution of the associated symbol-level description. But there are no more general rules.

The nature of reflective theories (what are the contents of the meta-components)

The tasks of the meta-project use knowledge-level models of the object-level project and thus manipulate the same objects as the designer: sets of tasks, models, methods and relations. As it will be seen in the experiment reports of the next sections, most of the meta-level constructions are object-level independent in the sense that they may be applied to various cases. This is an effect of the vocabulary change mentioned previously: once at the meta-level, one talks about modelling and no more about its subject.

3 The MetaKit

This section is an introduction to the KresT design environment and to its meta-level extension, the MetaKit.

KresT is generally considered as an implementation of the componential methodology ([Steels, 1990], [Steels, 1992c]). Similarly, the MetaKit may be viewed as an implementation of the theory presented in the previous section. Both practical and theoretical thoughts were carried out in parallel, with the explicit aim to make knowledge-level modelling of reflective activities straightforward.

First, KresT is presented briefly, then the basics of the MetaKit. We shall end with some examples of meta-level designs.

3.1 KresT

KresT implements the kernel of the componential methodology and thus proposes to build knowledge-level descriptions out of *tasks* (what to do), *methods* (how to do it), and *models* (with which information flows). These components may be described more precisely with a knowledge-level description language which is not imposed by KresT itself but must be provided by a software extension (called application kit, or appkit). The description language used in the current project is based on feature structures ([Kay, 1979], [Kay, 1986], [Gazdar and Mellish, 1989]) and, for clarity reasons, will be considered as being part of KresT in the rest of this document.

Using that language, one may assign to each component some *attributes* which may in turn have (or not) a *value*. These values may be either atomic objects (symbols, numbers, strings) or feature structures (set of attribute/value pairs) themselves (figure 7). These attributes form the knowledge-level description of the components.

Models have a specially important attribute, named **content-form**, which is interpreted as the abstract structure of the model (eg number, set,...). This information is enough for KresT to build a symbol level representation of the models.

Components may also be connected together with different types of relations. According to the componential methodology, any component may be linked to any other one but KresT implements only *task-roles* which link tasks to models or other tasks, and *method-roles* which link methods to models or tasks.

$$
\begin{bmatrix}
\text{type} & \text{model} \\
\text{model}-\text{type} & \text{case}-\text{model} \\
\text{name} & \text{"possible symptoms"} \\
\text{content} - \text{form} & \begin{bmatrix}
\text{structured} & \text{yes} \\
\text{structure}-\text{type} & \text{collection} \\
\text{ordered} & \text{no} \\
\text{duplicates} & \text{no} \\
\text{elements} & \begin{bmatrix} \text{content}-\text{form} & \text{symbol} \end{bmatrix}
\end{bmatrix} \\
\text{content} - \text{type} & \ldots
\end{bmatrix}
$$

Figure 7: Example of a feature structure description in KresT

The description of tasks and methods mainly consists in the description of the relations between theses components and their neighbours.

- *Task-roles* relate tasks to models or other tasks.

  They may be of three types:

  1. *input* roles relate models to tasks reading them
  2. *output* roles relate tasks to the models they modify (write to).
  3. *subtask* roles relate tasks to their subtasks

  The description of a role (which type, which role-filler) is part of the description of the task. Note that KresT imposes to indicate the direction of the information flow when a model is connected to a task. Similarly, it is not possible to connect two tasks without indicating which one is the subtask. Things work as if task-roles were arrows, denoting either the information flow direction or the subtask hierarchy.

- The *task/method* relation is, in KresT, a very tight one. Each task is connected to exactly one method (which is consistent with the componential methodology) but also each method may perform only one task. The rationale for that constraint comes from the fact that the methodology is case oriented: methods should not be viewed as sorts of generic algorithms but as instances of algorithms, applied in a particular context, with particular input and output models [1].

[1] The concept of generic algorithm corresponds in KresT to *library methods* which are ready made knowledge level descriptions of methods, intended to be pasted into the descriptions of methods of projects, as descriptions of their algorithmic and data-flow parts

- *Method-roles* relate methods to models and tasks.

  Because of the particular relation between a task and its method, a potential method-role is created when a task-role is established. In practice, KresT automatically creates an unlabelled method-role for each task-role and, for the sake of simplicity, identifies those roles in its graphical interface (which may lead to some confusion). The basic properties of a method-role are:

  1. its name, which is used as a key to select it, and to represent it on the diagrams (as labels along the links).
  2. its type, which is inherited from the task-role.
  3. The constraints imposed on candidate fillers (constraints on the knowledge-level description of the role-fillers).
  4. The multiplicity of the role (whether it allows multiple fillers).

Descriptions of properties of roles appear both in the owner's description and in the filler's description (only partially).

Finally, components may be grouped in larger units of description: a *project* is a coherent set of related tasks, models and methods; the coherence generally comes from the modelled object (one agent, one context, one root task,...). A project may also be viewed as the knowledge-level description of a knowledge system.

A *fragment* (or chunk) is a portion of a project, extracted from it and saved apart (generally in a library). Fragments may be reused afterwards by importing them in new projects. KresT implements a sophisticated mechanism to copy and paste fragments from one project to another.

The interface of KresT allows to work on the feature structure description of individual components and on their relations. This may be done partly by editing the feature structure representation and partly through a graphical interface. This interface proposes a limited set of diagram types:

- a *task-structure-diagram* represents the task/subtask hierarchy (either locally or globally).

- a *task-model-dependency-diagram* represents task/model relations for one task, together with information about the method/model relation.

- a *subtask-model-dependency-diagram* displays in one and a same window the same relations but for all subtasks of a given task.

To avoid diagram proliferation in this document, a mixture of those types of diagrams will be used here: diagrams will report tasks, subtasks, and involved models. Indications about the methods will also be given, as labels near the tasks. Of course, all diagrams will be commented in the text. Figure 8 summarises the notations used for the components and their different types of relations.

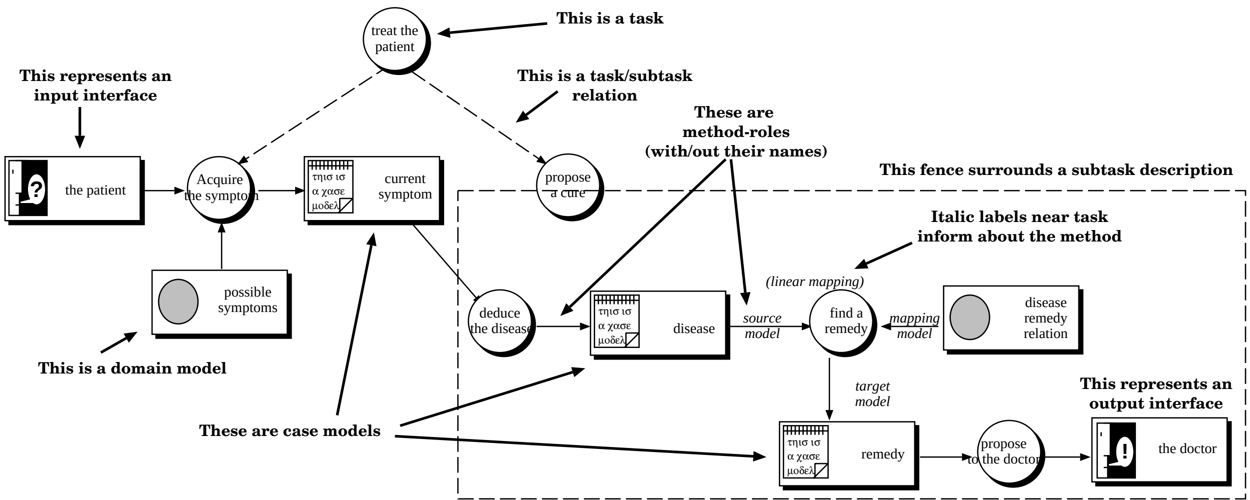

Figure 8: Graphical notation used in this document

More about KresT and its AppKits may be found in [Mc Intyre, 1993], [Goossens et al., 1993], [Steels, 1992c], [Steels, 1992a], [Jonckers et al., 1992].

3.2 What is the MetaKit

The MetaKit is an extension of KresT's standard Structured Basekit (version 1.7) which provides us with an ontology of meta-level design. More precisely, the MetaKit includes:

1. A set of methods and content-forms to design meta-projects at the knowledge level.

2. An implementation of those methods and content-forms so that meta-projects can be effectively encoded and run.

3. A re-implementation of some features of the Structured Basekit to allow reflection to take place.

4. A library of ready made fragments that may be reused in any meta-level project. Those are effectively potential building blocks for new knowledge-level designs, in which they may be imported, combined and modified.

3.3 Principles

The MetaKit follows the principles defined in section 2. This induces some additional terminology.

A *meta-project* is a project performing operations at the meta-level of another project, called the *object-project*. Meta-projects are *a priori* normal projects; they simply have the particularity to operate on a project, using methods and content forms defined in the MetaKit.

The MetaKit allows to work at each of the three sub-levels of the meta-level (knowledge, code and execution). This distinction is implemented by providing methods of the three types, not by forcing the meta-projects to operate completely at one of those sub-levels. As a consequence, tasks of the three types may be gathered into a single meta-project.

The tasks and methods of a meta-project operate on models of fragments of the object project. The smallest fragments are the components and their relations. The largest fragments are complete projects. Between those extremes, the MetaKit allows the full range of possibilities. The next section describes how such models are described.

These models acquire a value (the knowledge itself) either by reading a project file (project models) or by extracting parts of a project model with appropriate methods. Thanks to this mode of acquisition, there is no intrinsic dependency between the meta-project and the object-project and, as a consequence, the same meta-project may be successively or simultaneously working at a meta-level above several object-projects. This is why the MetaKit is particularly suited for generic meta-level operations.

According to the theory, there are no limitations imposed on the number of levels. Indeed, a meta-project may equally operate on a another meta-project, just seeing the latter as an ordinary object-project.

3.4 New model types for the meta-level

The case and domain models of a meta-project are models of parts of the object-project. The MetaKit provides a set of new possible values for the **content-form** attribute in order to describe such models.

The choice of the content-forms was done from a task perspective: the operations we want to perform determine the information we need. Thus, the content-forms, as a way to describe meta-level objects, have no pretention to universality but to usefulness, from an operational point of view.

All models belonging to the meta-level (ie referring to object-level objects) have the **meta** content form:

$$\begin{bmatrix}\text{content} - \text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & ... \\ \text{ref} - \text{level} & ...\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

The value of this attribute is in turn composed of two attribute-value pairs:

- the **ref-level** attribute indicates to which level the contents of the model belongs to, in the object-project (knowledge, code or execution).
- The type of the object-level object is in turn characterised by the value of the **referent** attribute, which can be, for example, **component** (task, model or method) or **project**.

Table 2 lists the possible configurations.

In the case the object is a component, its componential type must also be specified. In fact, **component** is not a value for the **object** attribute but an attribute itself which may have as value one of the symbols **task**, **model** or **method**. If the **component** attribute has no value, then the model may hold any sort of component.

For example, a model of the meta-project intended to hold a knowledge level model (of the object-project) has the content-form:

$$\begin{bmatrix}\text{content} - \text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \begin{bmatrix}\text{component} & \text{model}\end{bmatrix} \\ \text{ref} - \text{level} & \text{knowledge}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

In order to make a faithful model of the roles, the MetaKit has to explicitly implement the difference between method-roles and task-roles. This is why it complements KresT's role implementation by giving independent access to both types of roles and by dissociating the creation of method-roles

object	*level*	*describes:*
Component	Knowledge	The knowledge level description of a component
Component	Symbol	The symbol level description of a component
Component	Execution	The physical implementation of a component
Project	Knowledge	A knowledge level design
Project	Symbol	The source code of a program
Project	Execution	A loaded program (ie the computer resources involved in running the program)
Method-role	Knowledge	A method/model or method/task link. Two aspects are described: the definition of the role (from the method's view) and the state of the role (which fillers)
Encoder	Symbol	How to convert to the symbol-level (section 6.2)
Encoder	Execution	How symbol-level projects are loaded in memory (section 6.3)

Table 2: Basic meta-level content-forms

from the creation of task-roles. The possibility to create a method-role without a task role could appear as a source of errors in the basic KresT, but it is also an enrichment because some models may now be used by the method without concern for the task (eg model holding intermediate knowledge).

Because they hold a complex knowledge, method-roles may be explicitly modelled by meta-level models. Such models have as content form:

$$\begin{bmatrix}\mathrm{content-form} & \begin{bmatrix}\mathrm{meta} & \begin{bmatrix}\mathrm{referent} & \mathrm{Method-role} \\ \mathrm{ref-level} & \mathrm{knowledge}^2\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

On the other hand, the only information attached to a task-role is its type and its fillers and it may be retrieved by simply looking at the task. There is thus no real need for a specific content-form.

The **Encoder** related content forms are intended to model meta-symbol-level objects and will be explained in section 6.

Finally, one should note that there is (of course) no information about the particular object referred to by a meta-level model in its knowledge-level description. This information is the value of the meta-level model and thus belongs to its symbol level representation. The next section explains the basic ways to give a value to these models. A complete description of content-forms and their implementation may be found in appendix B

3.5 Acquisition of the meta models

The primary key to the object-level is the project. Once a model is filled with a knowledge-level project, any useful information may be retrieved by analysing it and specialised methods are provided which will be described thoughout this document (all methods are gathered in appendix A). The most basic ones are listed and commented here.

As a first step, knowledge level projects are acquired from the user (they could also be selected, for example, from a set, but anyway, this set must have been filled in by a user at some point). KresT projects are selected by their file names and need to be loaded in order to be inspected by the meta methods.

Note that it is a design decision (for the MetaKit) to view models as black-box data sources associated to information extracting methods. It has the advantage of hiding the difference between computed and stored properties of models (no redundancy problem, no design overload). On the other hand it also hides the model structure from the knowledge-level view.

- **Get model set** returns all models contained in a project
- **Get task set** returns all tasks contained in a project
- **Get method set** returns all methods contained in a project

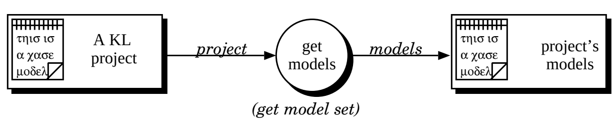

Figure 9: Extracting the models of a project

A similar approach is used to work with components, once extracted with those methods.

- **Get task subtasks** returns all subtasks of a task
- **Get task method** returns the method associated to a task
- **Get task input models** returns all models used as input-roles by a task
- **Get task output models** returns all models filling output task-roles
- **Get tasks reading model** finds all tasks which use a model as input
- **Get tasks writing models** finds all tasks which use a given model as output
- **Get method task** finds the task associated to a method

There is a complete set of methods to manipulate models describing method-roles. This set was built so as to allow all necessary manipulations of roles for verification purposes. This includes getting their type, their owner and their fillers and checking the associated constraints. Such models may be filled by analysing either owners (methods) or fillers (models). Note that subtask roles are not part of this scheme as they are more usefully covered by the task/subtask relation.

- **Get method roles** returns all roles of a method
- **Get role method** returns the method owning a role
- **Get role fillers** returns the set of components filling the role
- **Set role fillers** makes a model fill a role
- **Test if multiple role** test if multiple fillers are allowed for a role
- **Test if optional role** test if having no fillers should be considered as an error of that role
- **Test if input role** test if the role is an input one
- **Test if output role** test is the role is an output one
- **Test if subtask role** test if the filler must be a task
- **Test if possible filler** check a given model against the role's constraints
- **Fills roles** returns the set of roles that a given model fills

4 Verification of designs

As knowledge level designs grow, tracking inconsistencies becomes an increasingly tedious task. This task is usually decomposed in validation, verification and evaluation of the knowledge system [Ayel and Laurent, 1991], [Hoppe and Meseguer, 1991], [Meseguer, 1992], [Wielinga et al., 1993]. All are typical meta-level operations, usually performed by the designer and for which at least partial automation is possible.

This section focuses on *verification* of knowledge-level projects, this means checking a system against formalisable requirements. In particular, domain independent requirements, such as consistency and correctness, will be considered.

Problems encountered with knowledge design tools are not only a lack of consistency maintenance schemes, but also that such a maintenance is always eager (it may not be postponed until design completion) and, moreover, the concept of consistency depends often on the developer's style and methodology. The verification of a design may as well involve more than simply consistency checking:

1. compliance with some initial, formalised, requirements
2. completeness of knowledge level description of components
3. connectivity of the whole project

On the one hand, KresT offers some real-time consistency maintenance, mainly by interface restrictions and by propagating constraints through the method-roles, but this is not enough, eager and hard-coded.

On the other hand the MetaKit allows the knowledge engineer to design, at the knowledge level, his own verification strategies and to apply them to his projects[3]. All topics listed above may be covered and this section contains a series of examples of meta-projects performing detection and correction of particularly common cases:

- missing or incompletely described relations between models and tasks and/or methods (section 4.1).
- non-connectivity of the task tree (section 4.2).

[3] A previous experiment of verification, with KresT, but without the MetaKit, is described in [Cañamero et al., 1993]

- incompletely described components: methods (section 4.3), models (section 4.4) and tasks (section 4.5).

These concrete examples prove the success of the approach in the domain of verification:

1. the specifications of the object-level design are made explicit and clear;
2. this is done at the knowledge-level, by a decomposition of the verification task and a description of its subtasks;
3. such a description is enough to have an effective verification procedure running;
4. problems may be not only discovered, but also be fixed (to some extent);
5. the scheme may be extended to the more general case of automation of design (which is covered in the next section).

4.1 Connection of models

Common problems with models are:

- *unused*: a model used by no task. This may be due to cut links, forgotten models (eg if models were designed but not connected to tasks later on);
- *inactive*: a model used by no methods. This generally happens when a task has no associated method (in KresT: the task was not *stamped*), or when method roles were not assigned to models;
- *empty*: the model is not filler of any output role and thus never written. Not being used as output proves that if the program was run, the model would remain valueless;
- *useless*: the model is not filler of any input role and thus never read.

More problem types are discussed in the next subsections, from the point of view of tasks and methods.

The basic scheme for finding such problematic models is presented in figure 10. It may be modelled as a simple filtering task, keeping from the original set of models to check only the ones which are not used by any task. Note that the design is a bit simplified as the subtask **get tasks using model** is in fact decomposed into subtasks which look, in the object-project, for tasks using the current model as input and for tasks using it as input. The **tasks using model** model is the union of both sets. Refer to appendix A.1 for a description of the **filter** method.

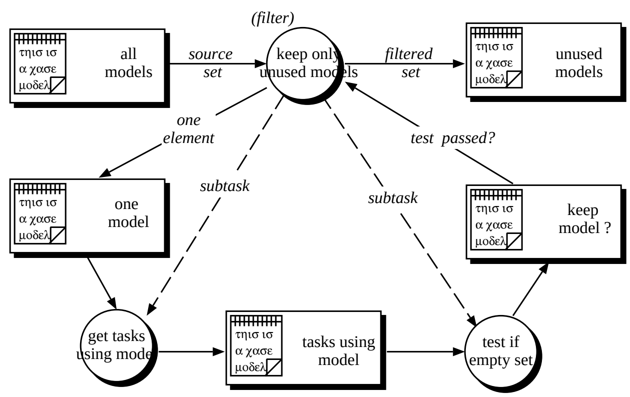

Figure 10: A sample design for finding stand-alone models

Once detected, such a model should be connected to some tasks (at least it should be filler of one input role and one output role). A dumb recursive machinery could be used to find acceptable tasks: try all possibilities and verify them (including other verification strategies). It is better to rely on method roles to have a finer grained model reconnecting strategy (see section 4.3).

To detect inactive models (connected to a task but to no methods), one may reuse the previous diagram and simply replace the **get tasks using model** task by the a single task, **get tasks using model**. The latter makes use of the **fills roles** library method (figure 11).

A safe way of looking for empty or useless model is to further analyse the method roles (rather than the task roles). This depends of course on the final goal of the modeller: if the project has to be encoded into an executable program, then *empty* and *useless* have a meaning only with respect to method roles. At the opposite, if the design is meant for analysis or documentation, then task roles have an equal (possibly greater) importance.

Repairing the problem of inactive models may be done in a rather mechanical way by checking all empty roles of the method attached to the task (if any). Figure 12 shows how method-roles may be tested for model com-

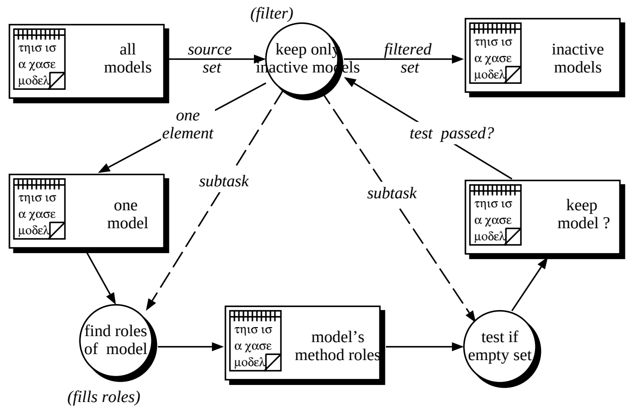

Figure 11: Finding an inactive model

pliance. The **model**, **task** and **method** models involved in that diagram are supposed to have been given a value by a preliminary phase of detection of inactive models (such as above), and thus hold an instance of a model connected to a task but not to the task's method. The main task has two non trivial subtasks. The first one, **is model compatible ?**, checks the compatibility of the model and the currently tested method role. It tests whether the model fulfils the constraints on the role's fillers, according to the method's knowledge level description; such a test is implemented as feature structure unification. The second subtask, **is task compatible ?**, has to verify that the task role and the currently tested method role have the same data flow direction.

Fixing the problem of empty models is strongly related to design completion techniques, which are the object of a subsequent chapter. A solution will be proposed in section 5.3.

4.2 Isolating task subtrees

The global task tree of a project is a directed graph. It should be connected when all tasks are subtasks of a root task, and thus all participate to the achievement of one unique goal. Nevertheless, a non connected task tree is not necessarily an error. It is sometimes the representation of multiple root tasks sharing the same knowledge, that is of several goals of one and a same agent. Detecting such cases is also useful at the time of encoding knowledge level designs into symbol level programs. Figure 13 presents a

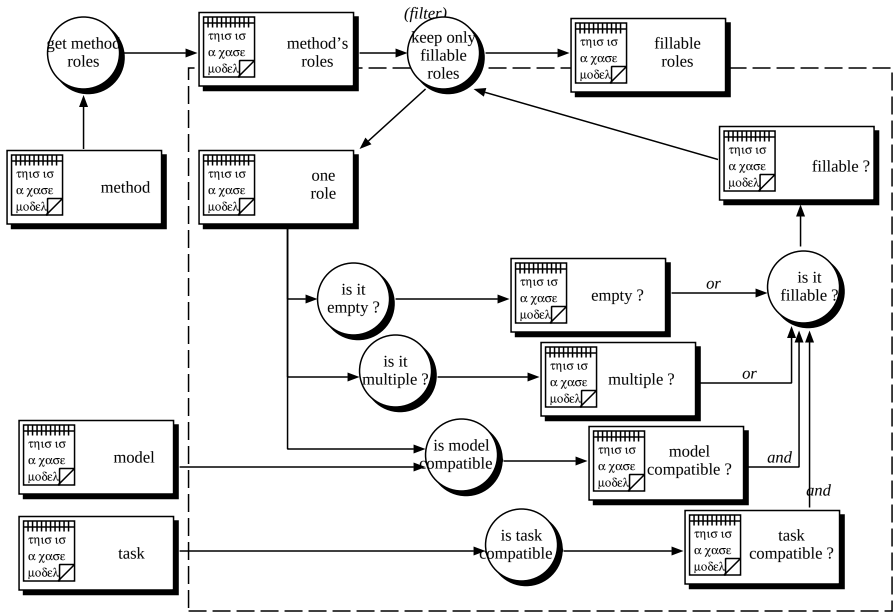

Figure 12: Finding possible roles for a model

way to identify the different connected subtrees of a task tree. The left hand part (the **find root tasks** task) has the goal to isolate tasks having no parent. The right hand part (the **build subtrees** task) classifies the rest of the tasks according to the root task they are descending from. The result is the collection of the subtrees.

The design in figure 13 hides the fact that in the second part, the task is searched recursively for all its subtasks, any level deep. This may be done nicely within KresT, as shown in figure 14.

4.3 Finding incomplete methods

A method is said to be incomplete when some of its roles have no fillers. This is why in the proposed solution (figure 15) the root task is further decomposed into:

1. gathering the roles associated to the methods to be verified.

2. eliminating those who have fillers.

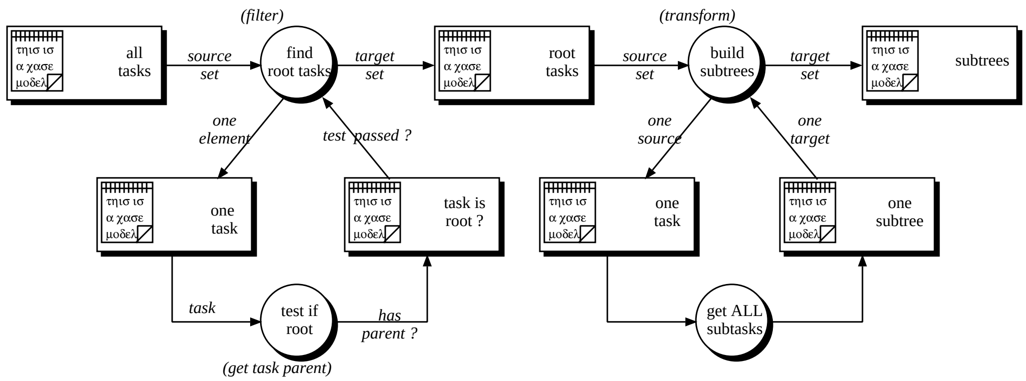

Figure 13: Building task subtrees

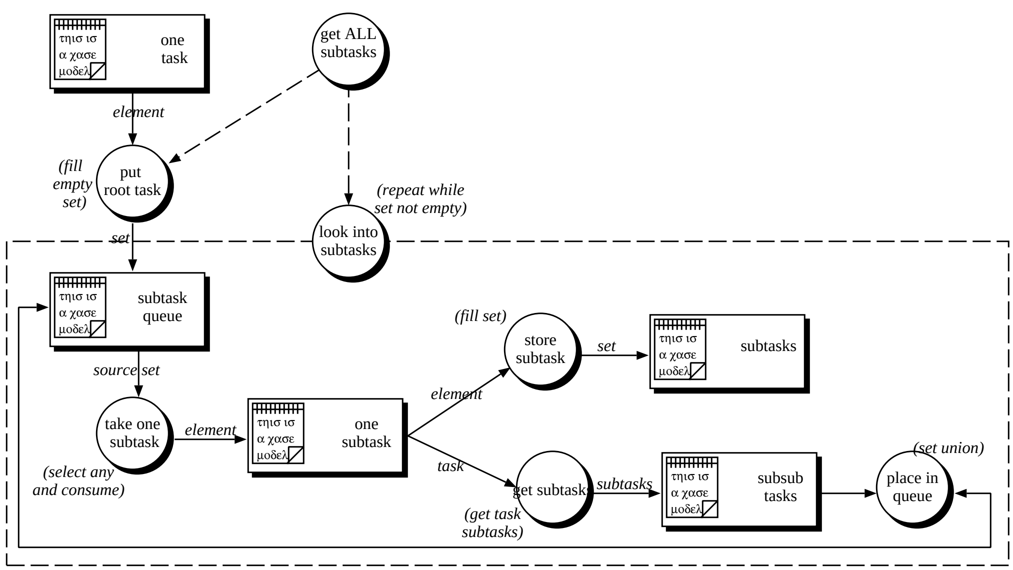

Figure 14: Recursively finding subtasks of a task

3. finding back the methods owning the remaining roles (the unfilled ones). This will produce a subset of the original set of methods.

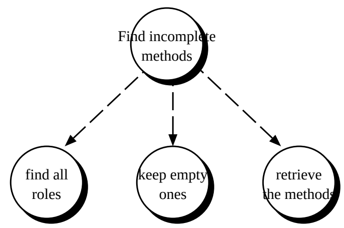

Figure 15: Task decomposition for finding incomplete methods

Gathering all method roles may be achieved by repeatedly taking the roles associated to the methods of the initial method set (left hand part of figure 16). As each time a set of roles is produced, then all of them have to be merged afterwards (right hand part).

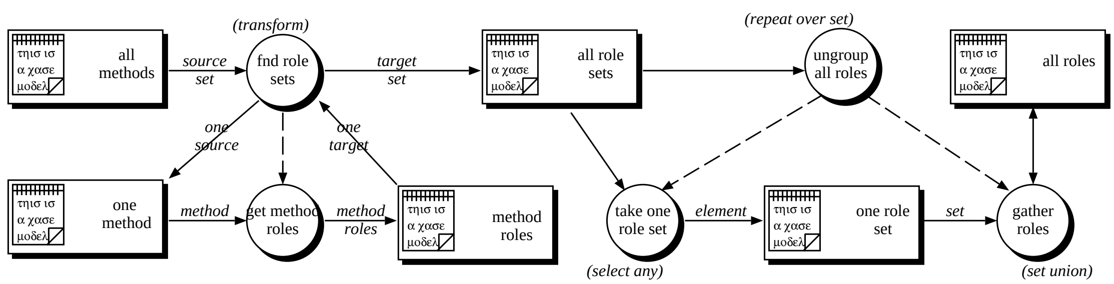

Figure 16: Gathering all method roles

Filtering out the roles having a non empty set of fillers is done by checking the emptiness of the role filler sets , as presented in figure 17. The right hand part of the same figure shows how the incomplete methods are retrieved from the roles. Note that since all models are described as sets, there is an automatic elimination of possible duplicates.

The problem of incomplete role assignment may be more complex if task roles are taken into account. For example, it is a common practice to connect important models to some task, for documentation purposes, and also to the subtasks which do effectively use them. In such a case, the method associated to the task will generally be one of the library's task decomposition methods and it will be effectively true that the model is not

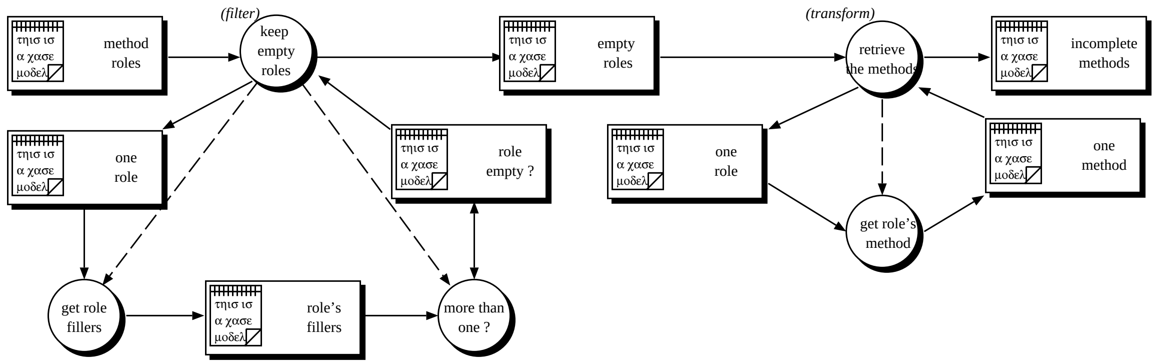

Figure 17: Filtering empty roles

used by the method of the main task. A method-role based verification strategy will fail and end up detecting an unassigned role. Fortunately, the MetaKit makes the difference between task and method roles (at the opposite, the basic mechanism of KresT leads to some confusion). Because the link between the model and the task is not reported as being a method role there is a nice way to refine the role verification strategy. If a task has an input (resp. output) model then the method of the task or of one of its subtasks must use it as an input (output) role. This may be implemented by replacing the test for membership of the method's input (resp. output) role filler. This is done by repeatedly testing all methods of the task subtree. The design of this strategy is a good example of reuse in KresT as it is mainly constructed as a variant of several already presented figures.

4.4 Completion of model descriptions

The main attribute of models is the **content form** attribute. It is, up to now, the only broadly used attribute, and furthermore the only one used by the KresT's encoding procedure to generate the code level representation. It is also the only one effectively used in the description of the method roles of the library methods (in the part concerning the constraints on the role fillers).

Normally, content forms of models are decided by the designer or by propagation of type constraints through the method roles. For example a selection method (selecting an element of a set) will impose that the input model is a sort of collection (ie a list or a set) and that the output model is of the same type as the elements of the set. If the verification of the method roles is performed beforehand and the corresponding repairs made, then, thanks to the constraint propagation mechanism of KresT, there will remain probably very few incomplete models. The remaining cases of possibly undetected incomplete descriptions are:

1. collections without further indications. For example, it is not stated if duplicates are allowed or whether the collection is ordered or not.

2. models not connected.

The first point may only be fixed using a default value (or by asking the designer); for the second point, refer to section 4.1.

If more attributes of models have to be given a value, then two tasks have to be performed: one is to detect incomplete models and the second is to assign a value to each attribute of those models.

The detection phase could, at a first glance, be accomplished by filtering out, from a set of models, the ones for which any of the required attributes has no value. The filter loop should thus have a subtask performing this test. This subtask could in turn be performed by filtering out from the set of required attributes the ones for which the currently tested model has a value. If the resulting set is not empty, then the model is faulty. Such a scheme is presented in figure 18.

The first problem comes from the fact that attributes may have structured values and thus, although this strategy effectively allows to detect the cases of attributes with no value, it will fail in the following other cases:

1. an attribute has a feature structure as value but this last is in turn incomplete. In the following example, the **content-form** attribute has a value but the **ordered** attribute should also have one.

$$\begin{bmatrix} \text{content} - \text{form} & \begin{bmatrix} \text{structured} & \text{yes} \\ \text{structure}-\text{type} & \text{collection} \\ \text{ordered} & \ldots \\ \text{elements} & \begin{bmatrix} \text{content} - \text{form} & \ldots \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

2. some attributes have to be checked only in concordance with some other ones. In the previous example, the **ordered** attribute must be checked only because the **structure-type** is **collection**.

3. some attributes are embedded in a sub-part of the description. In the example, the **content-form** attribute of the elements of the set also has to be filled in. This case will not be detected even if this attribute is member of the set of requested attributes.

4. inconsistencies between attribute values may certainly not be checked with that scheme. Take the case of a strictly ordered collection with duplicates.

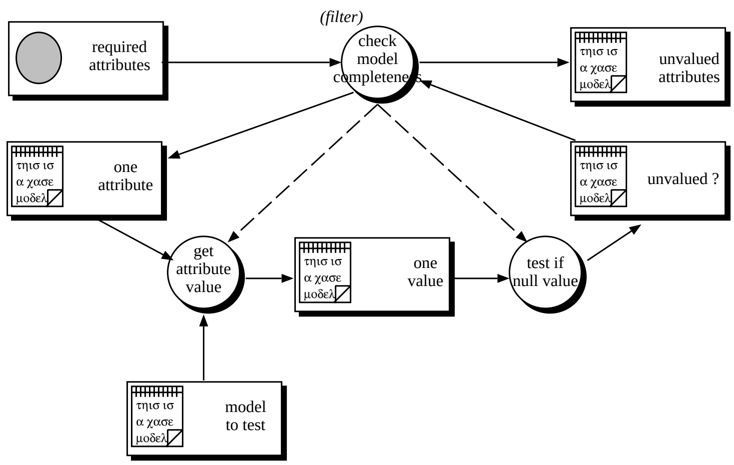

Figure 18: Testing a model for required attributes

There is thus a need for another solution. The one which is proposed now is based on the idea of model template. If a model is completely described (all required attributes have a value), then its feature structure description can be unified only with the description of models having the same attribute values, or no value for some attributes. So, assuming we have at our disposal a set of completely described models, then if more than one of them may be unified with the model being tested, we may deduce that this last must have at least one attribute with no value. Not only this principle may be used to build another method for detecting incomplete models, but also, by changing appropriately the contents of the reference set, other types of problems may be detected as well.

- attributes with no values may be detected by placing at least two models with different values in the set and rejecting models unifying with more than one of them;

- models with forbidden values will be pointed out by using a set of models with all possible legal values (or all forbidden ones) and rejecting models unifying with none of them (resp. any one of them);

- consistency between attribute values may be checked with the same technique: only legal combinations of different attribute/value pairs will be present;

- not only one attribute but more complex configurations may be matched simply by describing more precisely the models in the reference set.

This may seem to lead towards a gigantic reference set but, in practice, several small sets may be used, one for each topic to verify. In addition, the descriptions of the reference models need not to be complete themselves, only the covered attributes must have a value. For example, a set of models having only their **content form** attribute filled allows to detect problems with structure description while a set of models described only by a **content type** attribute could be used to test independently that attribute. If both set were merged, then the total number of cases would grow exponentially. On the other hand, illegal combinations may be removed from the set and thus detected by the procedure. Figure 19 shows how a single set may be used to detect illegal and incomplete models at once. The **set of templates** model is the reference set and is supposed to hold only complete and acknowledged descriptions.

Of course, this scheme may not be used when some attribute has an infinite set of possible values (numerical attributes, user free symbols..). In that case, the first solution should be used. Problems may also arise with recursive attributes (for example: sets of sets of ...) which may be considered to have an indefinite length value and, in that sense, the **content-form** attribute may fall into the category of infinitely valued attributes. The designer could accept some limitations to the depth of the recursion (in that case it is not so restrictive) or design another verification scheme to cover those cases.

A great advantage of this strategy is that it may be also used to assign values to attributes, simply by regarding templates as holding the default values of the attributes and by ordering them according to some sort of preference. Of course, this introduces some arbitrariness in the model descriptions, but it should be considered that if some attributes have no value, it is because the project is underconstrained. It also means that the designer

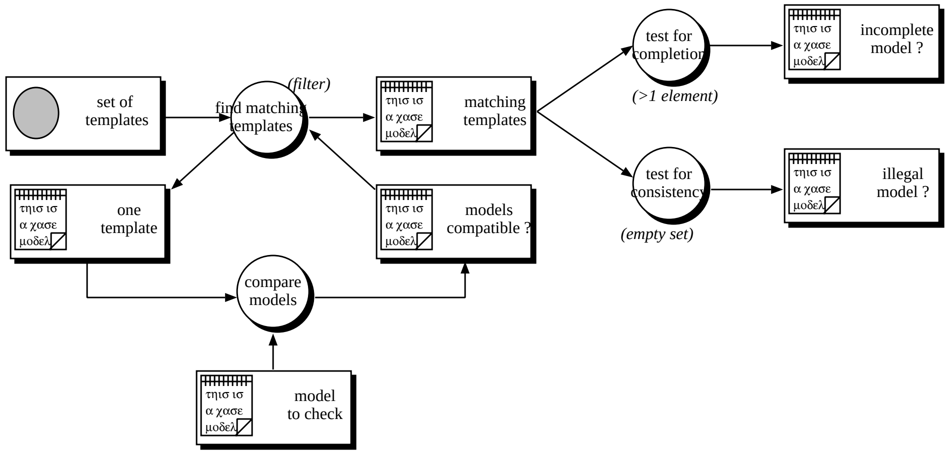

Figure 19: Searching for incomplete models

attached not enough importance to those attributes, possibly because they don't really matter with respect to his aims. Note also that the designer is free to implement or not that scheme or to have a report on whatever changes were made. Figure 20 shows how attribute filling may be designed in the case of a single template set. The **find matching templates** task is the same as in figure 19 and the model to check is supposed to have been checked for consistency.

Those two diagrams may easily be expanded to the case of multiple template sets and merged so that filling incomplete models occurs at the time of detection.

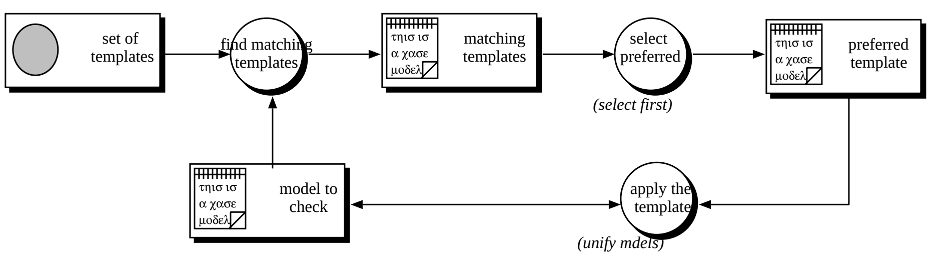

Figure 20: Fulfilling attributes with preferred values

4.5 When tasks have no methods

Let's now focus on the task/method relation. We shall say that a task has no method when it is not associated with any method. The reverse problem (having a task associated with several methods) is not considered to be possible, due to hard coded constraints in the design environment. In the case of KresT, a method (with an empty description) is systematically created together with any task. The only way to describe a method is to paste on it a description extracted from a library of ready made methods. The chosen library method is called the "method-type" of the task's method. Detecting incompleteness of the task/method relation thus amounts to looking for methods with no type. This may be easily done but is more difficult to fix.

When a library method has been selected and assigned to the task, then the task's method has new attributes: those which come from the library and, in particular, a **library-name** attribute reminding its origin. This is the attribute to check in order to detect "tasks with no methods". One could also look for role descriptions as an indicator of the method being effectively described. The precise strategy to detect such methods is exactly the same as the one which was used to detected incomplete models: just by looking for some specific attribute with no value.

Once detected, the most simple cases may be solved when the library is not too big. All methods of the library could be tried out and, each time, the modified task verified with respect to the models already connected (constraints on the role-fillers). This will work fine, for example, when there is a model of type **input interface** connected, as an input, to a task. Then only the acquisition methods of the library will be suitable as methods for that task. It is more a choice limitation strategy than a real fix of the problem. A more complex meta design is proposed in section 4.6, in the context of automatic design completion.

4.6 Case based verification and repair

The previous verification strategies were rather generic, in the sense that they could be applied to most projects of most designers; each example either improved some known strategy or produced a new one. But there is something that almost only the componential methodology together with the feature structure representation allows to perform safely and very easily: *verification with complete chunks*. It is much like making the system learn the designer's style and remember previous mistakes and choices at the time of verification or correction of subsequent projects.

In section 4.4, the idea of comparing models with elements of a predefined set of (in)correct models was introduced. If this scheme is extended to larger chunks of designs then it becomes possible to protect the projects against some specific erroneous situations. Additionally, the use of a set of templates to fill in the incomplete attributes may also be extended to complete fragments. Such an extension is less straightforward and requires the use of more and more complex domain models. These models are basically rules describing how to complete designs. More precisely, they are associations between incomplete designs and completion fragments, both being represented as sets of attribute/value pairs. A project to verify may be compared, by unification, with the incomplete designs. Similarly, the corresponding completion fragments may be added to the description of the current project, again using unification. This simplicity is the strength of the feature structure representation. Having relevant rule sets at disposal requires that the designer fills in domain models representing his know-how as he builds new projects. This effort will pay off in terms of efficiency and reliability.

As an example, we encountered previously the case of underspecified methods (section 4.5). The only possible solution was, at that time, to try all the methods from the library. Instead, using recorded cases allows to use a greater part of the method's environment for automatic completion of the method's description.

This type of knowledge modelling automation gives way to a bunch of applications, which make the object of section 5.

5 Automation of design

The previous section proposed a series of solutions to the problem of detecting faults in the knowledge-level descriptions of knowledge systems. Different ways to repair those faults were also presented and it is natural to wonder how far it is possible to increase the (re-)design power of the system.

This section reports on five experiments, mainly related to completion of design embryos from explicit rules. The principle was cursorily explained in section 4.6 and consists in (1) comparing the current project to some older fragments of designs and (2) deducing what has to be added to the project.

All these schemes require large fragment libraries and thus rely on good knowledge indexing and design documentation. These are weak aspects of KresT but are compensated by the fact that the componential methodology and the feature structure representation make reuse and sharing of knowledge simple and straightforward. These schemes also require extra work to build the libraries. But, firstly, this work may be spread over time because the library construction is incremental. Secondly, it pays off when libraries get big and give a lot of feedback. On the other hand, machine learning techniques could be used to improve and simplify it.

Although there are no universal and generic strategies for the design of knowledge-level projects, these experiments appear as a reusable basis which can be modified for particular needs.

The experiments are:

1. Completing with fragment libraries: the first possibility for design completion is to append fragments to selected parts of projects.

2. Completing with completion rules: explicit rule sets may help to tune completion (rules are taught to the system).

3. Specialised completion: a meta-project may model a project completion program.

4. Building projects from specifications: a meta-project may apply construction rules and design a complete project from scratch.

5. Testing design alternatives: a meta-project may produce several possible designs and evaluate them.

For the sake of clarity, some specific features of KresT's fragment library management shall be simplified by considering fragment libraries as sets of (partial) projects.

5.1 Completing with fragment libraries

The problem addressed here is the one of incomplete projects and the goal is that the system fills in the blanks in a project skeleton.

The diagram in figure 21 describes the task of finding suitable fragments in a library, given a set of incompletely described components (the **selected components** model). One fragment is then selected and used to complete the project.

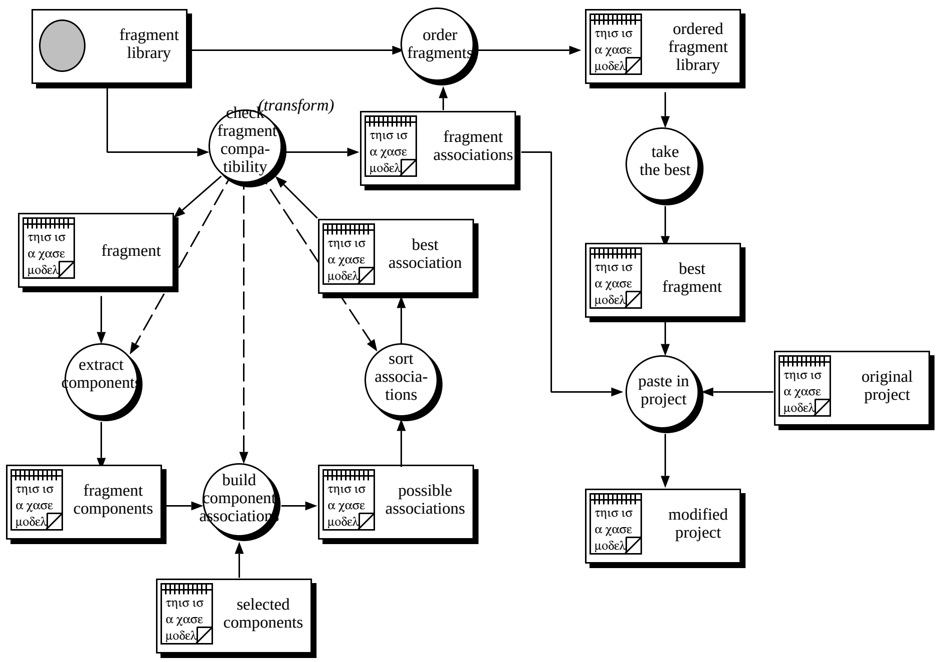

Figure 21: Browsing into a fragment library

The tasks are listed below, in execution order.

- **check fragment compatibility**

  The goal is to provide a list of fragments which are compatible with the current project and. The main information source is the **fragment library** model which holds a set of fragments. As mentioned above, fragments are themselves incomplete projects and thus the **content-form** attribute of this domain model is clearly:

$$\begin{bmatrix} \text{struct}-\text{type} & \text{collection} \\ \text{ordered} & \text{no} \\ \text{duplicates} & \text{no} \\ \text{elements} & \begin{bmatrix} \text{content}-\text{form} & \begin{bmatrix} \text{level} & \text{knowledge} \\ \text{object} & \text{project} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

In order to be really useful, the task has to transform the set of all fragments into a set of "fragment use guide". One thing is to know that a fragment can be added to a project, another is to know how the fragment components will be linked with the project's components. This is why each element of the **fragment associations** model should be a mapping from components of the current project to components of the candidate fragment (see the task **paste in project** below).

The **check fragment compatibility** task is further decomposed in:

- **extract components** retrieves the tasks, models and methods of the fragment (see section 3.5).
- **build component associations** finds the candidate relations for the current fragment. The **possible associations** model is thus of the same form as the **fragment associations** model.
- **sort associations** orders the previously found relations according to some criterion (number of matching attributes, relevance of matching attributes). Only the best one is kept and will be placed in the **fragment associations** model. If no association is accepted, the target model will be an empty relation.

• **order fragments** The gathered associations are then used as keys to compare and sort the fragments of the library. Note that incompatible fragments are automatically thrown away because they are associated to empty relations. The ordering predicate could be the same as for the **sort associations** task or it could complement it somehow. The choice of the criterion depends only on the personal vocabulary for component description that is used by the engineer and, unless they use the same attributes, several designers can not use the same criterion.

• **take the best** At this stage, selecting a fragment is an easy matter. If the fragment library could be correctly ordered, the first element of the **ordered fragment library** model should be selected. Otherwise, either the project is really under-documented or the library should be extended or the ordering criteria should be re-thought.

- **paste in project** The selected fragment is "pasted into" according to the linking informations contained in the **fragment associations** model (see above). Project components associated with fragment components will be unified (identified) with them, producing a description flow from the fragment to the project. All other fragment components are copied without modification to the project and componential relations (like roles) described in the fragment are transferred in the project as well.

As an example, such a scheme was used successfully to solve the problem of selecting a method for a task. At that time (section 4.5), the only solution was to try all methods of a method-library. Now, by placing the task into the **selected components** set, one may find templates that may be used for that task, respecting model connections as described in the object-project and possibly decomposing the task into subtasks.

5.2 Completing with completion rules

The previous subsection proposed a pure library-based solution for completion of designs, making the decision explicit at the meta-project level. The selection was made thanks to a more or less complex ordering predicate, comparing possible ways to incorporate a fragment at some selected place of the object project. If a development team plans to make a heavier use of fragment-libraries, then more guidance for fragment selection should be available. This may be achieved by pre-selecting fragment-libraries and/or fragments in a library, according to some properties of the project being built. Such a scheme should be used as a complement of the previous one.

Figure 22 shows a summary of the information flow occurring in the acquisition and execution phases of this pre-selection scheme.

The goal is to reduce the set of libraries browsed into at completion time. This is achieved thanks to the **completion rules** model which provides an informal classification of projects in terms of relevant libraries. As a relational model, it associates key properties of object-projects to fragment libraries. These project properties may be considered to be of the two following sorts.

1. explicit properties of the project which are expressed by project attributes. For example, if the **domain** is set to **technical-devices** then any library of fragments including models of technical-devices or methods for technical-devices becomes relevant.

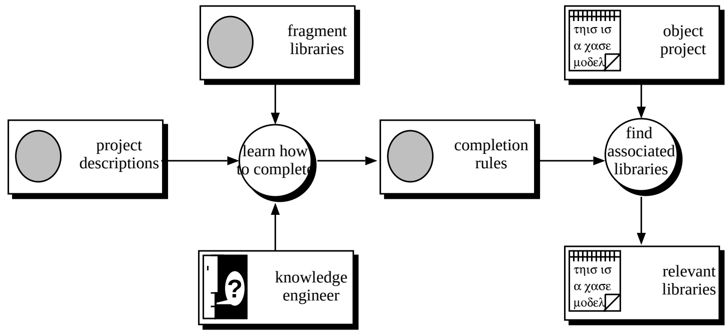

Figure 22: Acquiring completion knowledge and using it

2. emergent properties come from the fact that each project is a particular association of particular components. For example, if a project incorporates a diagnostic task (a task with **task-type** set to **diagnostic**), then the diagnostic related libraries should be enabled for the completion phase. A judicious set of tasks and model attributes may be found in [Steels, 1992b].

The **completion rules** model is thus a mapping model, more precisely a *description-to-class* model, according to the classification mentioned above. Because a given project may belong to several classes, this model should either hold a many-to-many relation (from sets of properties to set of libraries) or be considered as non-functional. This last solution has been adopted and thus, the model's elements are pairs formed of project descriptions on the one hand and fragment libraries on the other hand. Moreover, the project under development will have to be checked against all recorded project descriptions and the associated sets of libraries will all be triggered for the completion phase. This justifies the term of "completion rules".

In order to use a standard mapping function such as the **linear-mapping-with-alternatives** library-method, which will effectively produce all possible libraries, the **completion rules** model should be a set of associations from project templates (ie partial project descriptions, used as keys) to fragment libraries. The corresponding feature structure is represented in figure 23.

Such a model may be considered as a set of good design rules and shortcuts, and should be filled-in by the development team, as they build new projects and improve their design competence. Describing project properties is then sufficient to allow automatic and relevant completion of parts of

$$\left[\begin{array}{ll} \text{type} & \text{model} \\ \text{name} & \text{“completion rules''} \\ \text{content} - \text{form} & \left[\begin{array}{ll} \text{struct}-\text{type} & \text{collection} \\ \text{elements} & \left[\begin{array}{ll} \text{content} - \text{form} & \left[\begin{array}{ll} \text{struct}-\text{type} & \text{association} \\ \text{from} & (*) \\ \text{to} & (\dagger) \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

$$\left[\begin{array}{ll} (*) & \left[\begin{array}{ll} \text{content}-\text{form} & \left[\begin{array}{ll} \text{level} & \text{knowledge} \\ \text{object} & \text{project} \end{array}\right] \end{array}\right] \\ (\dagger) & \left[\begin{array}{ll} \text{content}-\text{form} & \left[\begin{array}{ll} \text{struct}-\text{type} & \text{collection} \\ \text{elements} & \left[\begin{array}{ll} \text{content}-\text{form} & \left[\begin{array}{ll} \text{level} & \text{knowledge} \\ \text{object} & \text{project} \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

Figure 23: Description of a rule set

subsequent projects. This requires the use of more project attributes (and *a fortiori* more fragment attributes) than in the basic version of KresT, these attributes, with their possible values, define *de facto* the developers ontology in knowledge-level modelling. In figure 22, the knowledge engineers are supposed to be the only information source for building the mapping model. How the **learn how to complete** task can be partially automated is an open issue.

An application of such a scheme may be found in specialised KBS development where few attributes of some components may be enough to decide to incorporate some common chunks. It may also be applied as a pre-filtering method before a more fine grained completion strategy.

5.3 Specialised completion

Let's start with an example: the automatic design of the knowledge acquisition part of a project. The job can be decomposed in the following subtasks (see figure 24).

1. **find or add acquisition root** has the goal to detect where the new acquisition tasks have to be placed in the global task tree. Figure 25 goes more into that task and shows how it is decomposed in:

    - **find all acquisition tasks** gathers all acquisition tasks of the project by examining their **task-type** attribute (which should then be set to **acquisition**).

- **find root tasks** extracts from the previous set the top most task.
- If there is no satisfying root acquisition task, then the **create root task** task will create one.

2. The **find empty models** task's goal is to identify the empty models (models not being output of any task, as defined in section 4.1). These models are the ones which are candidates to be connected to the output side of some acquisition task.

3. The **select the ones to acquire** task will filter those candidates, either according to some predicate or by asking the designer himelf. For example, it is good practice that domain-models are acquired from the expert (while most of case-models are computed).

4. The **add model acquisition** task may then complete the project with acquisition tasks, placed under the root acquisition task in the task tree. In the scheme proposed in figure 24, there will be one acquisition task for each model to be acquired, all based on a predefined acquisition template.

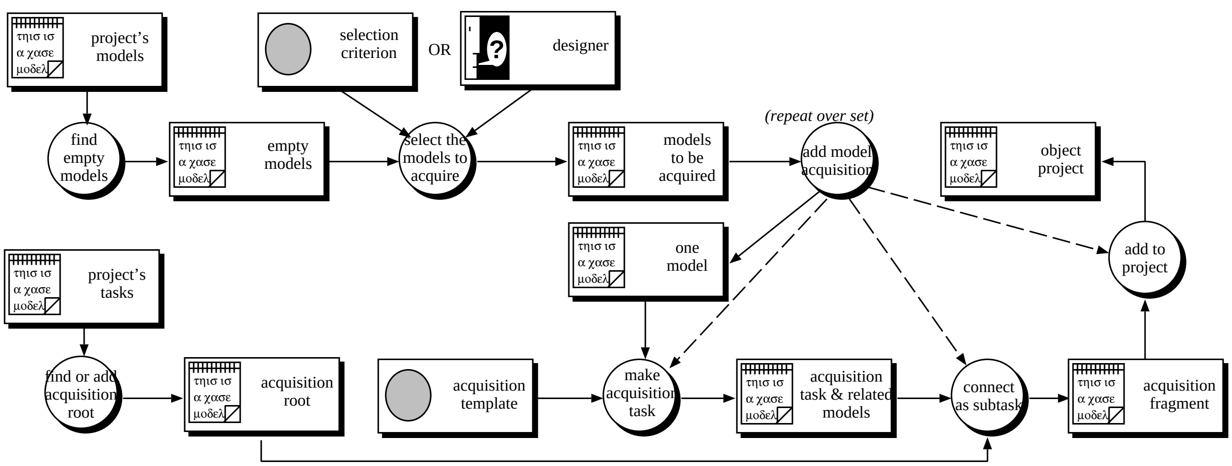

Figure 24: Adding acquisition tasks to a project

This is indeed a rather basic and rough strategy and may be refined by adding tasks performing a closer analysis of the object project. For example, model relations may indicate how to simplify the acquisition procedure

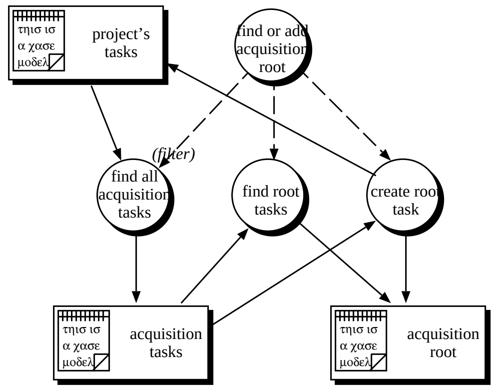

Figure 25: Finding the root task for acquisition

for the domain expert. Let's take the simple example of a linear mapping between diseases and cures (figure 26).

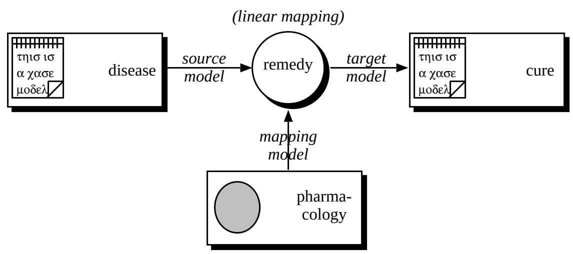

Figure 26: A sample use of linear mapping

In that case, the *source model* should obviously be member of the domain of the *mapping model*. This constraint may be used to select a particular method for the acquisition task of that model. On the other hand, if the source model was already acquired or computed from another domain model, then this last should be used as a source for acquiring the mapping model. Such type constraints appear in whole family of similar situations, making this example relatively generic.

This example proved that a specific design may be used to produce a customised automation of the design process. This not only goes further towards automation of the design process but also, because it is completely modelled at the knowledge-level, it does so in a generic and transparent way.

Compared to the previous method, this one gives more results but, on the other hand, requires more work. Both methods may be improved incrementally, in the sense that the way they are built allows to refine their knowledge and competence as the knowledge engineer's competence improves.

5.4 Building projects from specifications

This subsection presents an even more macro-like behaviour for automatic configuration of projects. The aim is to build the skeleton of a project, according to some specifications. Figure 27 shows the global diagram of the building task. It emphasises the fact that the knowledge is in the **fragment library index** model which maps the specifications to the fragments to be used to meet them.

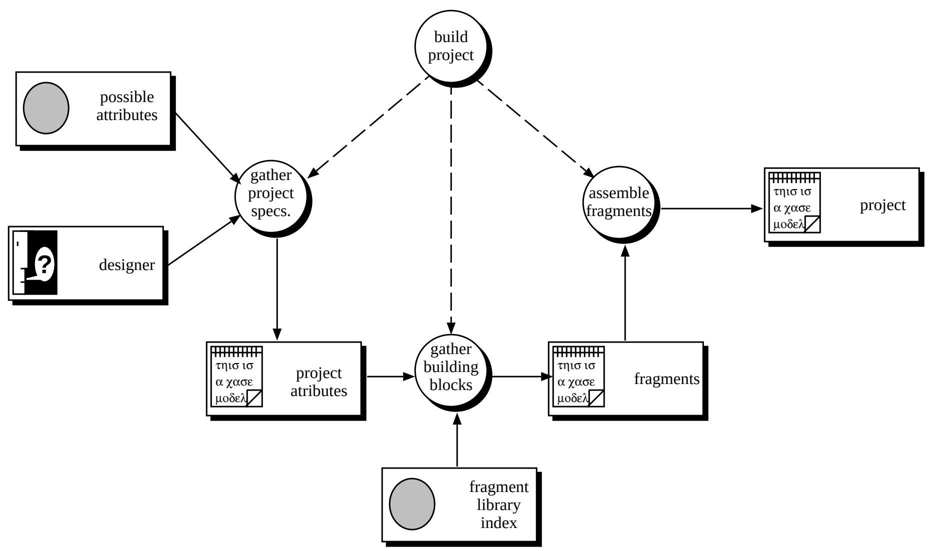

Figure 27: Building a project from scratch

This scheme was tested in the context of diagnosis applications. Several knowledge system specifications were built by assembling chunks, selected according to the value of project attributes such as the following ones.

- **inference-flow** will determine if forward or backward chaining will be used. Practically, this will select a complete configuration for such or such inference methods, including, for example, a *symptom-to-malfunction* model and associating the right method to the diagnosis task.

- **identify-faulty-part** will make the project include a functional model.
- **propose-remedies** indicates if a repair task (and its associated models) should be incorporated.
- **auto-acquisition** will trigger the automatic design of acquisition tasks

All those assembly operations are realised just by picking into a fragment library. This means that all the information concerning the interpretation of specifications is effectively visible in the **fragment library index** model.

The **gather project specs.** task may be realised simply by decomposing it into a simple acquisition task (select from a set) and a pre-configuration task which builds a project skeleton in which the only attributes are the ones selected by the previous method.

The other tasks are, from a technical point of view, similar to the ones encountered in the previous subsections.

Note that a strange property of this situation is that the meta-project is built before the object-project. This is perfectly consistent with the theory as once executed, the running instance of the meta-project will be effectively working on a description of an object-level project. The apparent temporal paradox is that the meta-project is *described* before the project is.

5.5 Testing design alternatives

It will of course happen that fragment selection fails to find a best fragment. The most common reason for that is the lack of selection criteria, which is in turn due to the fact that the final goal of the knowledge engineer may rarely be completely formalised. When this goal is to build a running knowledge system from the knowledge-level specifications, then the implementation of different candidates may be used to provide additional selection criteria. This will either eliminate non-working projects or, if still more than one works, will allow to compare project efficiency, precision and so on.

The diagram presented in figure 28 is intended to work in the situation of different competing fragments for a single project (gathered in the **possible completions** model). These fragments are each in turn added to the incomplete project, giving each time rise to a different completed project.

Each such project is then used as a specification to implement a knowledge system (the **encode** task, which builds the **program** model). Those systems will finally be compared using a set of benchmarks. In the proposed example, the comparison criterion is the validity of the system. Instead of comparing the results it produces to expected ones, one could measure the time and thus compare efficiency of the implementations.

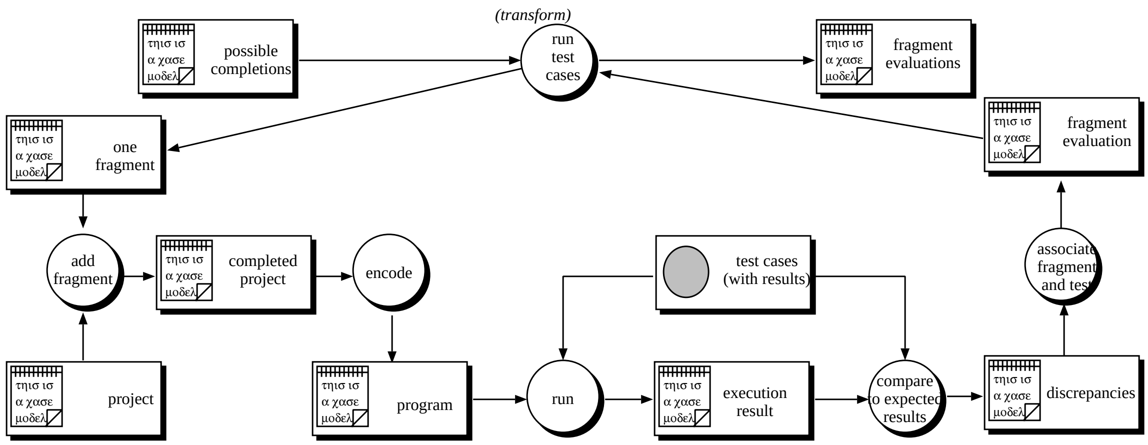

Figure 28: Evaluating a set of fragments for project completion

This diagram somehow anticipates on the results of section 6 which will show, with full explanations, how some automation may be achieved with respect to the symbol-level paradigm.

5.6 Conclusion

This ends off the part of this document involved in knowledge-level modelling alone. As emerged in the last subsection, the symbol-level now enters the arena. The next section deals with the modelling, at the knowledge-level, of symbol-level related manipulation.

The MetaKit proved to be useful and efficient in all tested configuration, from verification to repair of knowledge-level projects, and even to construction of brand new ones. This is thanks to a solid methodological ground (componential methodology) and a clear approach to the meta-level, preferring the clarity of separated levels to the mermaid song of self-reflection.

6 Controlling the symbol level

6.1 Introduction

In the previous sections, meta projects were mainly involved in analysing and modifying the knowledge-level aspects of a knowledge system. It is now time to see how evaluation of the symbol-level implementation of a knowledge-level design may in turn be described at the knowledge-level.

In tools like KresT, knowledge-level projects may be automatically interpreted in order to generate source code. This operation, known as *encoding* results in an executable piece of program (Lisp code in the current implementation), which is called the *code level* representation of the project. As stated before, loading this code into memory results in an *execution level* representation. Encoding is the only causal relation between code-level and execution-level descriptions, and it is a symmetric relation. In a MetaKit perspective, the encoding procedure is represented explicitely as the application of an *encoder* to a knowledge-level project. An encoder is a piece of code which may perform one of the following conversions:

1. encoding: from knowledge level descriptions (projects) to symbol level description (programs),
2. loading: from programs to memory,
3. decoding: from memory to programs (this is mainly used to produce an up to date description of the value of the models).

Since entire projects may be modelled as models having an appropriate content form, then encoding of a project may be viewed as a task in a meta project, using such models as input. Such a task may be performed by several methods, each corresponding to a different encoding scheme. The MetaKit provides ways to fully define the encoding/decoding operations from the knowledge level (section 6.2).

Controlling execution of a loaded program may appear difficult to describe consistently at the knowledge level. A scheme is proposed, based on a description of temporal and causal relations between components. This information is encoded into relations between symbol-level objects which are used to control their execution. This allows any form of flow control, including parallel execution (sections 6.3 to sec:flow-control). Moreover, some kind of automatic debugging may be envisaged (section 6.6). Between execution control and debugging, we will have a brief look on symbol-level evaluation (section 6.5).

6.2 Encoding to the symbol level

An important feature of the MetaKit is the ability to control how the knowledge-level description is to be encoded into a symbol-level description. A choice had to be made between two extremes: black-box encoding (easier to use, more transparent) and full representation of the encoder in the description of the meta projects.

The MetaKit introduces the concept of *encoder* as a content form. A model with such a content-form is intended to hold at least all the necessary information to encode from knowledge-level to symbol-level.

$$\begin{bmatrix} \mathrm{content-form} & \begin{bmatrix} \mathrm{meta} & \begin{bmatrix} \mathrm{referent} & \mathrm{encoder} \\ \mathrm{ref-level} & \mathrm{symbol} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Note that encoder models are described as being above the symbol-level but they are in fact between knowledge and symbol-level.

The MetaKit then proposes to do the encoding in two steps:

1. The components are described in a symbol-level language, one by one.
2. The collected descriptions are gathered and compiled into a complete symbol-level description in the target symbol-level language.

Figure 29 reports those two steps in a single diagram.

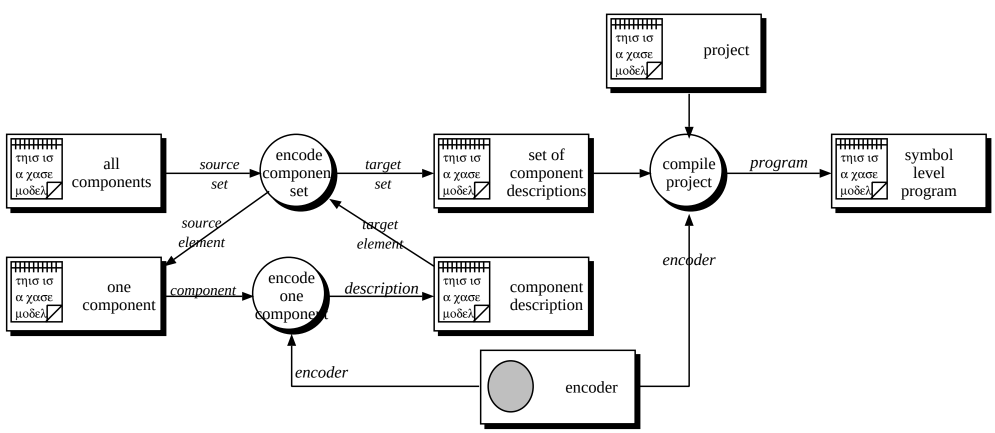

Figure 29: Encoding components

Making encoding a two step procedure has the following consequences.

- encoding rules are explicit;

- one can add rules to take into account new knowledge-level properties of components;
- it is possible to improve the accuracy of the encoding with respect to some implementation constraint;
- the only black-box part is the compilation phase. This is not an important drawback because changing the target language is not an every day decision and requires anyway more computational work.
- individual component descriptions need not to have the same form as in the final program. An intermediate language may be used and thus, encoding rules may stay relatively simple compared to the complexity of the programming languages. The compiler will be in charge of headers, footers, and all sorts of encompassing things.
- Moreover, it is not generally true that a program is simply a list of component descriptions (only object oriented languages allow that). An experiment of encoding to a rule-based system ([Tadjer, 1993], see also [Steels, 1992b]) showed, among other things, that models could be encoded as instructions and not necessarily as data.

Let's now have a closer look on what is an *encoder*. The MetaKit proposes, according to the componential methodology, to answer that question by providing methods and defining models. These methods make explicit the operations an encoder is expected to help for and the models point out which information is useful for those operations.

There are three main uses of encoders:

1. Encoding component descriptions from knowledge-level to symbol-level;
2. Compiling component descriptions into a program, in the context of their project;
3. Indexing components in memory (section 6.4).

The basic **Encode component** library-method converts one component's knowledge-level description into a symbol-level one. This is done by extracting from the encoder model the *component encoding rules*, which form a set of associations between component templates and corresponding symbol level descriptions. The **Encode component** task in figure 29 does

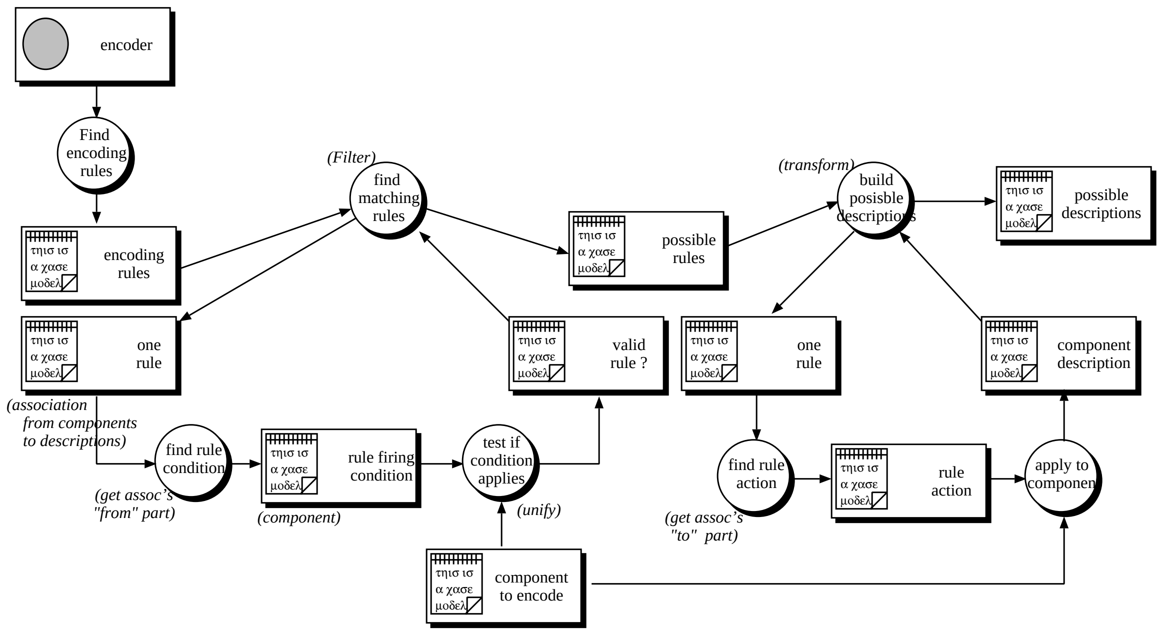

Figure 30: Decomposition of the basic encoding procedure

nothing else than matching those templates with the component to encode and returning the corresponding symbol level description.

All this means that individual encoding rules may be extracted from the encoder model, and then inspected or modified. The **content-form** attribute of the rule models should be as in figure 31.

$$\begin{bmatrix} \mathrm{structure-type} & \mathrm{association} \\ \mathrm{from} & \begin{bmatrix} \mathrm{content-form} & \begin{bmatrix} \mathrm{meta} & \begin{bmatrix} \mathrm{level} & \mathrm{knowledge} \\ \mathrm{object} & \begin{bmatrix} \mathrm{component} & \ldots \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \mathrm{to} & \begin{bmatrix} \mathrm{content-form} & \begin{bmatrix} \mathrm{meta} & \begin{bmatrix} \mathrm{level} & \mathrm{symbol} \\ \mathrm{object} & \begin{bmatrix} \mathrm{component} & \ldots \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 31: Content-form of encoding rules

The “from” part of those associations are partial knowledge-level descriptions of components which are used to trigger the rules: if a component may be unified with the “from” part of the rule, then this last may be applied. The “to” part of the encoding rules are the corresponding symbol-level descriptions. These are generic in the sense that they still need to be filled-in with specific attribute values of the component being encoded. This completion of the symbol-level description is performed by the **apply rule to component** library method (figure 30) which evaluates the symbol-level description in the context of the knowledge-level component, so that references to attribute values are replaced by the effective values at the time of encoding. As an example, the standard encoder defined in the MetaKit includes associations such as:

$$\begin{bmatrix} \text{type} & \text{model} \\ \text{name} & \ldots \\ \text{content} - \text{form} & \text{number} \end{bmatrix} \longrightarrow \begin{array}{l} \texttt{(make-model} \\ \quad\texttt{:name (value-of '(name))} \\ \quad\texttt{:tpstruct '(cl-number))} \end{array}$$

Defining new content forms and new attributes being the most simple way to improve KresT's expressive power, it is normal that the encoding of these content-forms and attributes was made transparent.

As shown in figure 32, the MetaKit handling of the meta-level covers both the meta-knowledge-level and the meta-symbol-level.

The final step of encoding consists in gathering all component descriptions into a single program. The way this is done depends on the target language. For example, encoding to CLOS code, as does the Structured Basekit, requires no further operations while encoding to languages such as C/C++ requires embedding of descriptions in global declarations. Attributes of the project itself could be used to tune the compilation phase, that's why the project plays a role with respect to the **compile project** library method. The compiling procedure itself is described in the encoder model (upper-left part of figure 33.

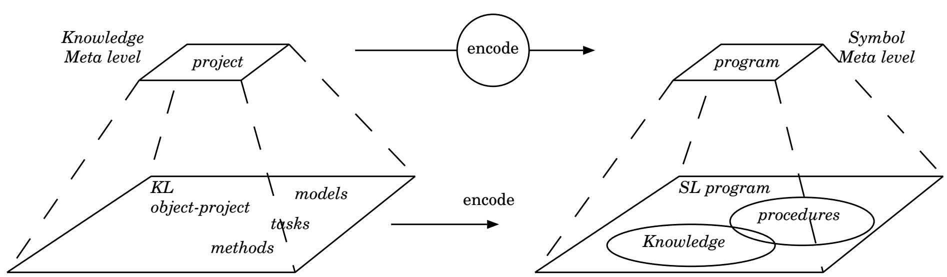

Figure 32: From knowledge-level to symbol-level

6.3 Working with symbol-level objects

Let's focus now on the use of code-level descriptions for the implementation of the object-level projects. In this section, the causal relation between code and execution level is made explicit. The execution-level will be studied in more details in the next section.

Programs are viewed as models of the meta-project, in which they have the following **content-form** attribute.

$$\begin{bmatrix}\text{content} - \text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \text{project} \\ \text{ref} - \text{level} & \text{symbol}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Once a symbol-level program has been generated, it may be used mainly:

1. to be exported as a stand-alone symbol-level description of the project

2. to be loaded into memory and to be run

The MetaKit incorporates methods to perform both operations. As the first is trivial, we will study only the second one and its consequences.

Once loaded, a program becomes a portion of a computer resources (memory and CPU time), this is its execution-level representation. The **Load-program** library-method takes in charge this precise point: load and remember where. It produces a model holding a sort of "load address" which represents the knowledge associated to the loading operation, that is which parts of the resources were used, for which purpose, where is stored the knowledge associated to such or such component. For such a model, the **content-form** attribute should be set as follows.

$$\begin{bmatrix}\text{content} - \text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \text{project} \\ \text{ref} - \text{level} & \text{execution}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

This model is expected to provide the meta-project with enough information for the following operations.

1. Run a task and check whether its goal was reached or not.

2. View the value of a model (note: this will be the value of the object-level model, we are thus operating at the meta-execution-level).

3. Backtrack the causes of an undesired state of the resources, check for failed tasks, empty models...

4. Decide, upon analysis of the state of the resources, which task has to be executed.

5. Run the entire program with different test cases.

6. Change the knowledge-level project so as to fix the faults detected in the execution-level.

Performing these operation requires to set up a relation between the knowledge-level description of the components and their execution-level representation and this may be done thanks to code-level descriptions. Given a component and its code, it is relatively simple to browse the memory to retrieve the current state of the associated execution object. In other words, the code-level descriptions serve as indexes in the resources, having an interface role between knowledge-level descriptions and execution-level resources. This is a positive side effect of encoding component by component.

This introduces the execution-level counterpart of the concept of encoder, namely the *loader*. There is a parallel between models with encoder-form, which were models of particular types of conversions from knowledge-level to symbol-level, and models with loader-form, which are models of how the code is interpreted to produce behaviour (execution). The similarity of the roles is explicit in the feature structure description of loaders:

$$\begin{bmatrix}\text{content}-\text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \text{encoder}\\ \text{ref}-\text{level} & \text{execution}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Figure 33 shows how encoding and loading of a project may be related to encoding and retrieving a component.

Note that the portion of resources retrieved as the execution-level aspect of the knowledge-level component needs not to be of a homogeneous type: models may become parts of code (for example user interfaces), while tasks may be encoded as data (eg specification of goals). But this scheme allows nicely to analyse some of their properties. This is the subject of the next section.

6.4 Working with execution-level objects

6.4.1 The problem

If such an effort has been put in identification of execution-level objects, it is to reach the objectives listed above. The next point to clarify is how properties of execution-level objects may be talked about at the knowledge-level.

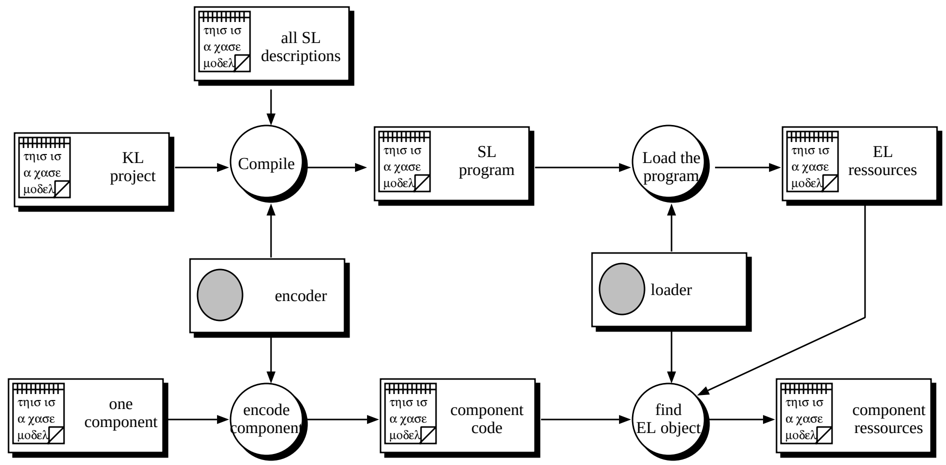

Figure 33: Global and local encoding

For example, it is important to know whether a task succeeded or not. However, the concept of success is task dependent and should be defined for each task. As such type of specifications concern states of the execution-level, it seems difficult to incorporate them in the knowledge-level descriptions.

Let's start with an example, in the context of the task/subtask relation. In the componential methodology, this relation is not, as often mistaken, similar to having subroutines of a main program[4]. Tasks are not programs, they simply make goals and sub-goals explicit. Several tasks thus represent several goals which are all pursued, not necessarily sequentially. Consequently, the subtasks of these tasks may be assigned to the agent in an *a priori* random way. This may be illustrated by the sample task decomposition presented in figure 34. Although knowledge acquisition and problem solving are presented as separate tasks, it is clear that the acquisition task will not be performed once for all. On the contrary, the expert using this KBS will switch to one of the subtasks of the **knowledge acquisition** task when necessary, even if he is already performing a subtask of **application**. So, gathering tasks under a parent task indicates that they all participate to a common goal, nothing more.

The MetaKit's solution consists in defining a vocabulary for describing execution-level related concepts with specific attributes and relying on the

[4] Up to now, KresT implements subtasks as the sufficient and necessary steps of a task's execution.

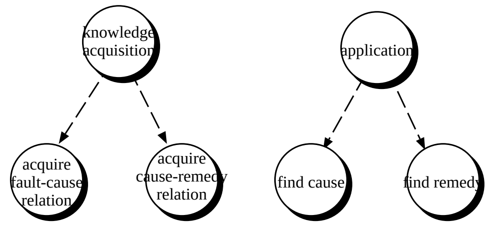

Figure 34: A sample diagnosis task structure

encoder to build their symbol-level implementation.

Firstly, it should be noted that part of the necessary information is already available, in an implicit form. For example, in the case of knowledge acquisition, the analysis of the task-model relations indicates which acquisition task should be run when problem solving fails due to lack of information.

But often, more will have to be said about tasks in order to have them executed at the right time. Indeed, the subtask selection may depend on the agent's environment as, in practice, tasks may not be executed at any time and are not always relevant. This means that an ontology of the domain must be defined in order to be able to talk, at the knowledge-level, of the relations between the state of the tasks and the state of the models (the environment). Therefore, models should also be described more precisely, and again, this knowledge-level ontology should be transformed in execution-level predicates by the encoder and the loader.

This involves additional attributes for tasks, which will serve the encoder to build the right program. These new attributes are detailed in the next section.

6.4.2 An ontology for flow control

This section presents an application of the ideas presented above to the specific problem of control of task execution. As briefly stated before, the effective order of task execution at run-time will depend on specific task attributes. One could think that such attributes should not be part of a knowledge-level description because they concern the execution-level. In fact, these attributes should simply describe other aspects of the tasks and, on the other hand, are not more related to the symbol-level than any other attribute used by the encoder. It is a matter of choosing the right vocabulary. This is what is going to be done now, step by step.

When the system is running a program, the first step towards selecting the next task to run is to find out which tasks are executable now, in the current state of the environment. In other words, there must be a predicate, say **executable-p**, which could be applied meaningfully to every task. The most common reasons preventing a task to pass the test are the following ones. Such reasons are called the *pre-conditions* of the task.

- effective sub-tasking: the task may be executed only during parent task execution.

  Example: Subtasks of a problem-solving task, most of subtasks in the diagrams presented in this document.

- explicit control-flow: the task must be executed only after another one.

  Example: In figure 34 above, search for a remedy may not occur before identification of problem's cause.

- model access: input models are not readable and/or output models are locked (defined below)

  Example: In the same example, the task may not be performed if no cause has been identified by the previous task.

- method pre-conditions: the environment does not permit the execution of the associated method.

  Example: See below.

Similarly, some situations may enforce the executability of a task:

- model fix: the task's goal is to fill in a model when it is empty, more generally to fix any problem with model access.

  Example: Failure of a task to find the cause of a device malfunction makes the acquisition task of the symptom-cause mapping model eager to be executed.

- control-flow: this task has to be executed before another one, which, for some reason, is expected to be performed.

  Example: As above.

These two lists lead to the definition of new attributes for tasks (listed in table 3) which are to be placed in the feature structure description of the tasks, as sub-attributes of the **conditions** attribute. Figure 35 shows the template for task descriptions, and figure 36 proposes two simplified examples.

attribute	*usage*
conditions	root attribute
execution	conditions to make the task executable
success	conditions which make the task successful
after	which tasks must have been executed before
during	which tasks must be currently running
empty	set-models which must (not) be empty
changed	models which must have changed value since last time

Table 3: Attributes describing task control

The only logical connective between conditions is conjunction. Although all attributes have a negated form, this is a strong limitation, which is simply due only to the experimental nature of this description language. The current language proved the feasibility and usefulness of the approach but will be completely reworked for the release version of the MetaKit. On the other hand, enriching the encoder so it accepts more complex descriptions is not really a complex task.

$$\begin{bmatrix} \text{type} & \text{task} \\ \text{name} & \ldots \\ \text{conditions} & \begin{bmatrix} \text{execution} & \ldots \\ \text{success} & \ldots \end{bmatrix} \end{bmatrix}$$

Figure 35: Template of a task description

6.4.3 Encoding

For such descriptions to be useful for the control of execution, they must be translated to the symbol-level. This may be performed either by enriching the encoding rules in order to handle those cases, or by using a design

An acquisition task:

$$\begin{bmatrix} \text{execution} & \begin{bmatrix} \text{empty} & my-domain-model \end{bmatrix} \\ \text{success} & \begin{bmatrix} \text{changed} & my-domain-model \end{bmatrix} \end{bmatrix}$$

A repair task:

$$\begin{bmatrix} \text{execution} & \begin{bmatrix} \text{succeeded} & find-cause \\ \text{changed} & cause-model \end{bmatrix} \\ \text{success} & \begin{bmatrix} \text{changed} & remedy-model \end{bmatrix} \end{bmatrix}$$

Figure 36: Simplified examples of task conditions

completion phase before the encoding phase. The experiment was done with an enriched encoder, which proved the feasibility of the scheme, but also its limitations. What should remain from the current implementation are the basic encoding rules. For example, the **changed** condition is encoded by (1) adding code to save the model's value at entry of task and (2) adding to the **executable-p** predicate associated with the task a piece of code comparing the value of the model with the previously saved value.

The next step will probably be to use a two stage encoding, using the techniques of sections 5.2 and 5.3 so as to add "execution control" tasks to the project. The meta-project (more precisely, its encoding part) would then have to build an "extension" of the object project, based on the following facts.

- Deciding to run an acquisition task because a model is empty or a method failed means that the model may be tested for emptiness and the method for failure and that corresponding predicates must exist.
- Those predicates may not be used directly by the meta-level project because the execution-level of the meta-project is the object-project. As an example, the method **test for empty set** may not be applied to a meta-model representing a set-model of the object project because it will test for the emptiness of the meta-model (which is not a set anyway).
- In fact, the predicates may be requested from the meta-level, but in the sense that the meta-project will decide that an object-level test must occur. The explicit meta operation is thus to run a task, the goal of which being to find out if that model is empty or not.

So, theoretically, the prerequisite for performing such tests is to design additional tasks in the object project. In other words, this part of the duty is related to design completion, as described in sections 5.2 and 5.3. Of course, such modifications of the object-project should be temporary to avoid covering the designer's project with execution related fragments.

6.4.4 Compatibility with KresT

KresT users have for the moment only one way to describe execution control, it is to use task decomposition methods from the library. Many examples were encountered in the diagrams of the previous sections. In particular, these methods implement sequential subtask execution, cliches like filter and transform, or repetition over set elements. They all impose that the only tasks which may be executed in concordance with a given task are its subtasks. For example, there is no way to retry knowledge acquisition once the application tasks started. This limitation is compensated by the simplicity of the projects. But, this is, as mentioned above, not enough to cover real cases. It should be stressed that the MetaKit's way of expressing control is still compatible with this restricted scheme. These are example of how control specification may be rewritten:

- An equivalent for **sequential task execution** is to describe the subtasks with:

$$\begin{bmatrix} \text{during} & \mathbf{parent-task} \\ \text{after} & \mathbf{previous-subtask} \\ \text{before} & \mathbf{next-subtask} \end{bmatrix}$$

- For **repeat while set non empty**, the parent task should contain a condition like:

$$\begin{bmatrix} \text{non}-\text{empty} & \mathbf{source-set} \end{bmatrix}$$

It is also possible to mix the old style of design with the new one. But, as there may be only one execution-level system, the task-decomposition methods of the library should be encoded as if the new scheme was used. This may be done very simply by adding rules to the encoder (eg as above).

6.4.5 Extension to model and method ontologies

The description of task conditions relies heavily on description of states of models. In the MetaKit, the model terminology is completely fixed, and conditions referring to models depend upon the content-form of the model. For example, sets come with predicates like **empty** and **singleton**, numbers may be compared to zero and boolean models may be true or false. Thus, the encoder knows how to transform conditions of the tasks to tests of object-level models.

The temptation is great to apply to models the terminology scheme we applied on tasks. This has not been implemented yet, but it appears that tasks could use more model independent conditions. For example accessibility of models is not a relevant concept in stand-alone machines, limited to keyboard, screen and memory resources (this explains mainly the weakness of the implementation for model state description). But if input/output interfaces could be files, remote machines or input/output devices, such aspects should be more developed and made of course not only hardware independent, but also symbol-level independent. In that case, be “model independent” means that concepts like “being (un)accessible” could be described, in knowledge-level terms, within each model. More generally, the semantics of any model attribute could possibly be redefined for each model. As a consequence, model’s properties could be requested as conditions of tasks or methods without the task (or method) knowing what it practically means for such or such model.

The method related terminology has its own particularities, not related to tasks and models. Besides concepts similar to task conditions, methods also have “return values”. Selecting a method for a task implies the definition of possible final states of the algorithm. For example, all the current library-methods end up with a symbol denoting how successful was the method. The notion of success here has nothing to do with goals, as for tasks. The success of a method is measured by the algorithm itself. It is thus an internal, hidden criterion and never comes from external specifications.

6.4.6 Conclusion

The intention with flow control experiments was to clear the ground, both at theoretical and practical level. This is done, and, although the preference would have gone to more conclusive examples and applications, the door is open.

From all this, the general rule which emerges, about such terminologies, is that the existence of the terms (properties) is stated at the meta-level, their definitions are local to each component, and their meaning is made explicit by the encoder.

6.5 Test sessions

Once running, the knowledge system may be validated and evaluated. That is compared to requirements and to concurrent systems. From the point of view of the componential methodology, those two different issues come down to asserting properties of the execution-level.

The MetaKit allows to build handsome designs of evaluation procedures. One of them is presented here, which uses training sets to evaluate projects. It is decomposed in the following steps:

1. acquire the input/output training set
2. run the program with the training set
3. compare with the requirements, analyse the state of the components

This may be considered as combininig two types of evaluation: first, the general principle of filling the inputs and watching the outputs, then, a case approach comparing the global state of the result to the root task's goal. How they both fit in the meta-level scheme is summarised in figure 37.

The project preparation phase which appears in figure 37 consists in looking for input and output interface models in the knowledge-level project so as to be able to artificially fill the models with the values listed in the **training set** model. The latter is viewed as an association from a set of input values and a set of output values. The elements of those sets are themselves associations between models and model values (figure 38).

Note that preparation also includes disabling the acquisition and presentation tasks, using the methods of the previous section (addition of attributes preventing tasks from being executed).

The comparison between requirements and effective output is straightforward: manipulating associations and sets are basic operations and it is not difficult to produce the set of discrepancies (eg a list of ill-valued models).

If this scheme is coupled with the one of the previous section, it becomes possible to check the effective state of resources and watch if task goals have been reached, according to their knowledge-level specifications. For

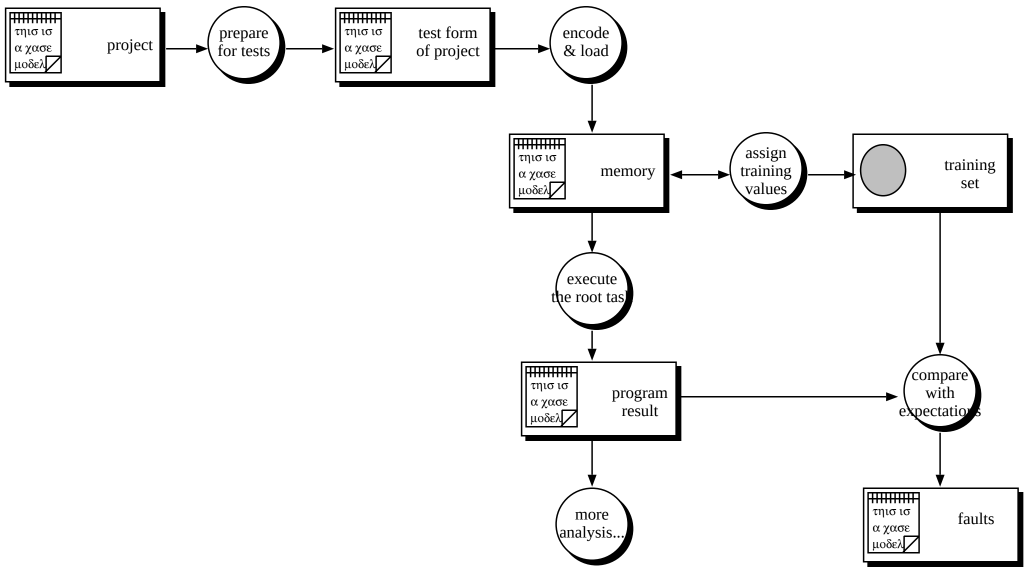

Figure 37: From the project to its evaluation

$$\begin{bmatrix} \text{struct}-\text{type} & \text{collection} \\ \text{ordered} & \text{no} \\ \text{uniform} & \text{yes} \\ \text{duplicates} & \text{yes} \\ \text{elements} & \begin{bmatrix} \text{content}-\text{form} & \begin{bmatrix} \text{struct}-\text{type} & \text{collection} \\ \text{ordered} & \text{no} \\ \text{uniform} & \text{yes} \\ \text{duplicates} & \text{no} \\ \text{elements} & \begin{bmatrix} \text{content}-\text{form} & (\textbf{see below}) \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} \text{struct}-\text{type} & \text{association} \\ \text{from} & \begin{bmatrix} \text{content}-\text{form} & \begin{bmatrix} \text{meta} & \begin{bmatrix} \text{object} & \begin{bmatrix} \text{component} & \text{model} \end{bmatrix} \\ \text{level} & \text{knowledge} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{to} & \begin{bmatrix} \text{content}-\text{form} & \begin{bmatrix} \text{meta} & \begin{bmatrix} \text{object} & \begin{bmatrix} \text{component} & \text{model} \end{bmatrix} \\ \text{level} & \text{execution} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 38: Elements of the training set

example, one could check for failed tasks, then look for their non-executed subtasks, and deduce the reasons of the failure.

Again, this is possible thanks to the MetaKit but has been tested only on small projects. No methodology or generic scheme has been drawn yet from the experience.

Another application of the proposed solution is to use this validation mechanism during the design process: in section 5.5 the problem was that several possible completions of a project were competing. The evaluation scheme may be applied to several projects successively and the result used to rank candidates. The quality of the selection depends on the analysis criteria for the execution-level results.

6.6 Automatic debugging

The point here is to show how the same tools and techniques could be used with a different view on design.

In the previous evaluation modes, the project had to be tested against some specifications. In the context of debugging, the project is supposed to be at least approximately correct but subject to failure in some, still unknown, cases. What is then expected from the meta-project is a partial redesign of the knowledge-level project so as to produce a better program. This is simpler than it may sound. For example, failed tasks may be easily retrieved (as in the previous section) and, using the goal specifications, and/or the associated method's return value, the system may decide to try-out different replacements for the involved parts. The effective behaviour will be a sort of design-run cycle. It is much like having the meta-project supervising the program execution and taking back control when things go wrong. This intervention would happen only when needed, and only to fix one problem at a time.

In addition, if the object-project may be always run under control of the meta-project, then the latter may also record the history of the object program and build, time after time, a training set which can be used for further designs or repairs.

There is a real challenge when there is not a predefined part of the meta-project to fix the problem, for example when the possible problem types have not been identified beforehand. In that case, a meta-project may be used to build a second meta-project the goal of which will be to find a solution. The complete power of automatic design may be used to apply different meta-project designs on the same problematic object-project. This could be done just by using the various techniques proposed along this document. It is not more complicated to work at the meta-level of a meta-project than at the meta-level of an object project, according to the theory, it is the same thing, and the same tools may be used.

6.7 Conclusion and outlook

The work on the symbol-level, in relation with its knowledge-level, appears to be a promising issue in KBS tool design, especially when, as it is done here, reflective strategies may be expressed at the knowledge-level, in the same formalism as the object system.

The next step is clearly now to gather more experiments, in order to define general rules for writing meta-projects, and to identify reusable fragments and generic methods as building blocks for such designs.

7 Conclusion

It seems to me that this study succeeded in proving that knowledge-level reflection is not an abstract dream of AI researchers but, thanks to a clear theory, may effectively be used to build better knowledge modelling environments, and, with those, to make more reliable designs. The proposed approach and resulting tools applied successfully to many relevant problems, encountered in the everyday work of knowledge engineers.

However, the symbol-level related meta-projects should be now studied in more details and the proposed schemes extended and applied to larger and more realistic cases. Making explicit description of the logic of task execution is as important as having verified projects. This is an open field for knowledge-level reflection.

Thanks

I wish to thank for their kind support Samia Beziou, Sabine Geldof, Luc Goossens, Alain Petit, Luc Steels, Viktor Tadjer, Walter van de Velde and all members of the VUB AI Lab.

A List of meta level methods

This section gathers the descriptions of all methods referred to in the previous parts of this document. Trivial functions of the underlying basekit are not reported.

The description of the methods are of the form:

Method name [*Type of method*]

role name	type of role	constraint on fillers
example-role	input	set of symbols

- explanation of the methods procedure.

The type of the role will be one of *input*, *output* or *subtask*. An additional *multiple* postfix indicates that several models (or tasks) may fill the role.

The constraint on the role fillers shall be are documented briefly instead of reporting on the complete feature structure each time. In particular, *model*, *task* and *method* shall be used as shorthands to denote models with respective content forms model, task and method (see appendix B).

A.1 Basekit methods

The Structured Basekit was enriched with a few generic methods.

Filter [*Basekit method*]

Source set	input	set
One element	output	element of*Source set*
Test passed?	input	boolean
Target set	output	same type as *Source set*
Subtask	subtask/multiple	

- This methods allows to select some elements of a set according to a predicate.

  For each element of *Source set*, put its value in *One element*, then execute all the subtasks sequentially; after execution, watch the value of the boolean model filling the role *Test passed ?* and, if it is `true` then add the element to the set *Target set*.

Transform [*Basekit method*]

Source set	input	set
One source	output	element of *Source set*
One target	input	element of *Target set*
Target set	output	set
Subtask	subtask	

- The aim of the method is to allow one to apply repeatedly some tasks to all elements of a set, and to collect the result each time.

  Take, one after each other all the elements of *Source set*; each time put the value in the model filling the role *One source*, then execute all the subtasks and, finally, add the value of the model filling the role *One target* to the set *Target set*. The subtasks are responsible for filling each time the value of *One target*.

A.2 Methods working with models of models

This section lists the methods of the MetaKit itself which were used in the diagrams of this document.

Get tasks reading model [*MetaKit method*]

Model	input	model
Tasks	output	set of tasks

- Fills the model *Tasks* with the set of the tasks using the value of *model* as input.

Get tasks writing model [*MetaKit method*]

Model	input	model
Tasks	output	set of tasks

- Fills the model *Tasks* with the set of the tasks using the value of *model* as output.

A.3 Methods operating on models of methods

Get method type [*MetaKit method*]

Method	input	method
Type	output	symbol

- Return the identification of the library method used for the method *Method*

Set method type [*MetaKit method*]

Method	input	method
Type	input	symbol
Typed method	output	method

- Paste the library method identified by *Method type* on the knowledge-level method *Method*. The result will be the value of the *Typed method* filler. Note that this is simply done by unification.

Get method task [*MetaKit method*]

Method	input	method
Task	output	task

- Sets the value of *Task* to the knowledge-level task associated to the value of *Method*

Test model compatibility [*MetaKit method*]

Model	input/multiple	model
Compatible?	output	boolean

- Checks (by unification) if all models have compatible attribute values:
  - models have the same value for the attribute
  - one model has a value and the other ones don't
  - models have attribute value pairs as values; and those are in turn compatible.

A.4 Methods operating on models of tasks

Get task input models [*MetaKit method*]

Task	input	task
Input models	output	set of models

- Fills the model *Input models* with the set of the models the task *task* uses as inputs.

Get task output models [*MetaKit method*]

Task	input	task
Output models	output	set of models

- Looks for the set of models used as output by the task *Task* and places them in the model *Output models*

Get task method [*MetaKit method*]

Task	input	task
Method	output	method

- Sets the value of *method* to the knowledge-level method associated to the value of *Task*.

Get task subtasks [*MetaKit method*]

Task	input	task
Subtasks	output	set of tasks

- Fills the model *Subtasks* with the set of the substaks of *Task*.

A.5 Methods operating on roles

Fills roles [*MetaKit method*]

Model	input	model
Roles	output	set of method-roles

- Fills the model *Roles* with the set of the method-roles having the value of *model* as filler.

Get method roles [*MetaKit method*]

Method	input	method
Roles	output	set of method roles

- Retrieves all method-roles defined for the method *Method* and places them in the model *Roles*

Get role method [*MetaKit method*]

Role	input	method-role
Method	output	method

- Finds the method owning the method-role *Role* and places it in the model *Method*

Get role fillers [*MetaKit method*]

Role	input	method-role
Fillers	output	set of components

- Fills the model *Fillers* with the set of the fillers of the method-role *Role*.

Add role filler [*MetaKit method*]

Role	input	method-role
Component	input	component
Filled role	output	method-role

- Appends the model *Component* to the list of the role fillers of the method-role *Role*

Test if multiple role [*MetaKit method*]

Role	input	method-role
Multiple?	output	boolean

- The model *Input?* will receive the value `true` if the method-role *Role* is a multiple role (allows multiple fillers).

Test if optional role [*MetaKit method*]

Role	input	method-role
Optional?	output	boolean

- The model *Input?* will receive the value `true` if the method-role *Role* is an optional role.

Test if input role [*MetaKit method*]

Role	input	method-role
Input?	output	boolean

- The model *Input?* will receive the value `true` if the method-role *Role* is an input role.

Test if output role [*MetaKit method*]

Role	input	method-role
Output?	output	boolean

- The model *Output?* will receive the value `true` if the method-role *Role* is an output role.

Test if subtask role [*MetaKit method*]

Role	input	method-role
Subtask?	output	boolean

- The model *Subtask?* will receive the value `true` if the method-role *Role* is a subtask role (ie its filler must be a task).

Test if possible filler [*MetaKit method*]

Role	input	method-role
Component	input	component
Match?	output	boolean

- Gives to the model *Match?* the value `true` if the value of *Component* matches the constraints on the role fillers of *Role*.

A.6 Methods operating on attribute representation

Get attribute value [*MetaKit method*]

Description	input/multiple	project
Attribute	input	symbol
Value	output	project

- This method fills in the *Value* model with the set attribute/value pairs found as value of the *Attribute* attribute in the description. This method takes advantage of the identification between projects and their feature structure representation.

Models unify? [*MetaKit method*]

Model	input/multiple	model
compatible?	output	boolean

- Ths method checks if the descriptions of models filling the role *Model* may or not be unified and sets the value of the *Compatible?* model accordingly.

Unify fragments [*MetaKit method*]

Fragment	input/multiple	project
Unified fragment	output	project

- This method does exactly the same as *Unify models* but works on whole fragments instead.

Fragments unify? [*MetaKit method*]

Fragment	input/multiple	project
compatible?	output	boolean

- See method *Models unify?*.

A.7 Methods operating on models of projects and fragments

Extract model set [*MetaKit method*]

Project	input	project
Model set	output	set of models

- Fill the model *Model set* with the list of the models defined in the value of *Project*

Extract task set [*MetaKit method*]

Project	input	project
Model set	output	set of models

- Fill the model *Model set* with the list of the tasks defined in the value of *Project*

Extract method set [*MetaKit method*]

Project	input	project
Model set	output	set of models

- Fill the model *Model set* with the list of the methods defined in the value of *Project*

A.8 Methods for encoding

Encode component [*MetaKit method*]

Component	input	k.l. component
Encoder	input	encoder
Code	output	s.l. component

- This method looks in *encoder* to find an encoding rule matching the description of *component*; rhis rule is applied, resulting in a symbol-level description which is placed into *Description*.

Compile project [*MetaKit method*]

Project	input	k.l. project
Descriptions	input	set of s.l. components
Encoder	input	encoder
Program	output	s.l. project

- This method fills the model *program* with the result of the application of the compiler associated to *encoder* to both the set of symbol level descriptions found in *Descriptions* and the knowledge-level project *project*.

Export program [*MetaKit method*]

Program	input	s.l. project
Output Interface	output	symbol

- This method copies the value of *Program* to a file named by the symbol *Output Interface*; KresT allows only keyboard/screen interfaces, so the file system is somehow hacked by using symbols for filenames.

B Description of meta content forms

And their implementation

All meta objects have the same kind of content forms :

$$\begin{bmatrix}\text{content}-\text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \ldots\\ \text{ref}-\text{level} & \ldots\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

where the `level` attribute indicates whether the model holds a knowledge / symbol / execution level description of the `object`; which can be a component (either task, model or method) or a project.

KL Projects The content form for models holding complete projects; from such models, task, model and method sets may be extracted.

$$\begin{bmatrix}\text{content}-\text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \text{project}\\ \text{ref}-\text{level} & \text{knowledge}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Models The content form for models holding knowledge level models of the object project.

$$\begin{bmatrix}\text{content}-\text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \begin{bmatrix}\text{component} & \text{model}\end{bmatrix}\\ \text{ref}-\text{level} & \text{knowledge}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Tasks Same for tasks

$$\begin{bmatrix}\text{content}-\text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \begin{bmatrix}\text{component} & \text{task}\end{bmatrix}\\ \text{ref}-\text{level} & \text{knowledge}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Method Same for methods

$$\begin{bmatrix}\text{content}-\text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \begin{bmatrix}\text{component} & \text{method}\end{bmatrix}\\ \text{ref}-\text{level} & \text{knowledge}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Method role The links between models and methods

$$\begin{bmatrix}\text{content}-\text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \text{method}-\text{role}\\ \text{ref}-\text{level} & \text{knowledge}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Program sources Models having this content form hold encoded programs, that is symbol level projects.

$$\begin{bmatrix}\text{content} - \text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \text{project}\\ \text{ref} - \text{level} & \text{symbol}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Code for components The code associated to individual tasks, models and methods

$$\begin{bmatrix}\text{content} - \text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \begin{bmatrix}\text{component} & ...\end{bmatrix}\\ \text{ref} - \text{level} & \text{symbol}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

Encoders The specifications of an encoder.

$$\begin{bmatrix}\text{content} - \text{form} & \text{encoder}\end{bmatrix}$$

Run time objects The execution level of components and projects

$$\begin{bmatrix}\text{content} - \text{form} & \begin{bmatrix}\text{meta} & \begin{bmatrix}\text{referent} & \begin{bmatrix}\text{component} & ...\end{bmatrix}\\ \text{ref} - \text{level} & \text{execution}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

References

[Ayel and Laurent, 1991] Ayel, M. and Laurent, J. P. (1991). *Validation, verification and Test of Knowledge Based Systems.* John Wiley & Sons, Chichester, England.

[Cañamero et al., 1993] Cañamero, D., Geldof, S., and McIntyre, A. (1993). Coupling modeling and validation in COMMET. In Meseguer, P. (Ed.). , *Proceedings of EUROVAV '93.*

[Carle, 1992] Carle, P. (1992). *Mering IV: un langage d'acteurs pour l'intelligence artificielle distribuée intégrant objets et agents par réflexivité compilatoire.* PhD thesis, Laforia - Université Paris 6.

[Ferber, 1988] Ferber, J. (1988). Reflection in actor languages as a result of an equivalence between entities and functions. In *Meta-Level Architectures and Reflection*, Amsterdam. North-Holland.

[Gazdar and Mellish, 1989] Gazdar, G. and Mellish, C. (1989). *NLP in LISP*, chapter Chap. 7 - features and the lexicon, pp. 215–282. Addison Wesley Publishing Co.

[Gödel, 1931] Gödel, K. (1931). Uber formal unentscheidbare satze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 173–198.

[Goossens et al., 1993] Goossens, L., Vroede, K. D., and Slodzian, A. (1993). *Structured Basekit 1.6 Notes.* VUB AI Lab.

[Hoppe and Meseguer, 1991] Hoppe, T. and Meseguer, P. (1991). On the terminology of VVT. In Jenkins, P. and Plaza, E. (Eds.). , *Proceedings of the European Workshop on the Verification and Validation of Knowledge Based Systems*, Jesus College, Cambridge, UK.

[Jonckers et al., 1992] Jonckers, V., Geldof, S., and Devroede, K. (1992). The COMMET methodology and workbench in practice. In *Proceedings of the 5th International Symposium on Artificial Intelligence*, 341–348, Cancun, Mexico.

[Kay, 1979] Kay, M. (1979). Functional grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistic Society.*

[Kay, 1986] Kay, M. (1986). Parsing in functional unification grammar. In Grosz, B., Spark Jones, K., and Lynn Webber, B. (Eds.). , *Readings in NLP*, 125–138. Morgan Kaufmann Publ. Inc., Los Altos, CA.

[Ladriere, 1957] Ladriere, J. (1957). *Les limitations internes des formalismes.* Nauwelaerts.

[Laird et al., 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33, 1–64.

[Maes, 1987] Maes, P. (1987). *Computational Reflection.* PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium. Also as VUB AI-Lab TR-87-02.

[Mc Intyre, 1993] Mc Intyre, A. (1993). *KresT 2.5 user manual.* Knowledge Technologies.

[Meseguer, 1992] Meseguer, P. (1992). *Validation of Multi-Level Rule-Based Expert Systems.* PhD thesis, Universitat Politecnica de Catalunya.

[Newell, 1982] Newell, A. (1982). The knowledge level. *Artificial Intelligence*, 18, 87–127.

[Newell, 1990] Newell, A. (1990). *Unified Theories of Cognition.* Harvard University Press.

[Putnam, 1989a] Putnam, H. (1989a). *Mind, Langage and Reality.* Cambridge University Press.

[Putnam, 1989b] Putnam, H. (1989b). *Realism and reason.* Cambridge University Press.

[Smith, 1984] Smith, B. C. (1984). Reflection and semantics in Lisp. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, 23–35, Salt Lake City, Utah. also: Xerox PARC Intelligent Systems Laboratory Technical Report ISL–5.

[Steels, 1990] Steels, L. (1990). Components of expertise. *AI Magazine*, 11(2), 29–49.

[Steels, 1992a] Steels, L. (1992a). Corporate knowledge management. In Cuena, J. (Ed.). , *Knowledge-oriented software design*, 91–116. North-Holland Pub. Co., Amsterdam.

[Steels, 1992b] Steels, L. (1992b). *Kennissystemen.* Addison Wesley, Reading, MA.

[Steels, 1992c] Steels, L. (1992c). Reusability and configuration of applications by non-programmers. AI Memo 92-4, AI-Lab, Vrije Universiteit Brussel, Brussels, Belgium.

[Tadjer, 1993] Tadjer, V. (1993). Procedural encoding of declarative knowledge in the environment of the krest workbench. Technical Report AI Memo 93-10, VUB AI Lab.

[Tarski, 1972] Tarski, A. (1972). *Logique, Semantique, Metamathematique 1923-1944.* Armand Colin.

[Van de Velde, 1993] Van de Velde, W. (1993). Issues in knowledge level modeling. In J.-M. David, J.-P. K. and Simmons, R. (Eds.). , *Second Generation Expert Systems.* Springer Verlag, Berlin.

[Wielinga et al., 1993] Wielinga, B., Akkermans, H., and Schreiber, G. (1993). Validation and verification of knowledge models. In Cardenosa, J. and P.Meseguer (Eds.). , *Proceedings of the European Symposium on the Validation and Verification of Knowledge Based Systems*, 29–50.

[Wielinga and van Harmelen, 1993] Wielinga, B. and van Harmelen, F. (1993). Knowledge-level reflection. Technical report, Universisty od Amsterdam.