The Knowledge Engineering Review, Vol. 11:3. 1996. 253–280

TFL: an algebraic language to specify the dynamic behaviour of knowledge-based systems

CHRISTINE PIERRET-GOLBREICH and XAVIER TALON

L.R.I., CNRS URA 410, Université Paris Sud, 91405 Orsay Cedex, France (Email): [pierret, talon] @lri.fr

Abstract

TFL, the Task Formal Language, has been developed for integrating the static and dynamic aspects of knowledge based systems. This paper focuses on the formal specification of dynamic behaviour. Although fundamental in knowledge based systems, strategic reasoning has been rather neglected until now by the existing formal specifications. Most languages were generally more focused on the domain and problem-solving knowledge specification than on the control. The formalisation presented here differs from previous ones in several aspects. First, a different representation of dynamic knowledge is proposed: TFL is based on Algebraic Data Types, as opposed to dynamic or temporal logic. Second, dynamic strategic reasoning is emphasised, whereas existing languages only offer to specify algorithmic control. Then, TFL does not only provide the specification of the problem-solving knowledge of the object system, but also of its strategic knowledge. Finally, the dynamic knowledge of the meta-system itself is also specified. Moreover, modularisation is another important feature of the presented language.

1 Introduction

Knowledge acquisition was traditionally viewed as an *extraction/transcription* process. An opposite view is generally adopted now: the *modelling* approach (Breuker and Van de Velde, 1994; Wielinga et al., 1992). In this process, a particular model is especially important: *the model of expertise* (ME), which is the result of the conceptualisation stage. This model is an abstract description of the problem-solving behaviour of an agent (human or artifact). Such a functional specification of a Knowledge Based System (KBS) is made at a conceptual level, and is independent of the implementation. Different methodologies (e.g. Role Limiting Methods (McDermott, 1988), Components of Expertise (Steels, 1990), Generic Tasks (Chandrasekoran, 1988), PROTEGE (Musen, 1989), KADS (Breuker & Van de Velde, 1994) and associated conceptual modelling languages have been proposed. They offer *informally* or *semi-formally* defined primitives to describe the ME. However, their lack of clear semantics limits the ME advantages, since it causes difficulties in its understanding, validation, reuse and implementation. Therefore, to support a more precise and *formal* specification of the ME, several languages have been more recently defined in knowledge-engineering, (e.g. $(ML)^2$, (Van Harmelen and Balder, 1992), K_{BS} SF (Jonker & Spee, 1992), ForKADS (Wetter & Schmidt, 1991), KARL (Fensel, 1995b), QIL (Aitken, et al., 1992) Momo (Linster et al., 1992) and DESIRE (Langevelde et al., 1992). Two groups are distinguished among them (Fensel & van Harmelen, 1994). Languages in the first group mainly aim at formalising the ME, in particular the KADS model of expertise. They have a denotational semantics. Languages of the second group aim at operationalising the ME. They have an operational semantics. However, the trouble is that most of those languages, whatever their type, do not deal with the *control problem* in a completely satisfactory way. For instance, most of them only consider *one single* problem-solving method to solve a task, which is generally statically defined in advance with a fixed control structure. Languages which are more flexible and allow one to describe

alternative problem-solving methods, yet provide no clear means of specifying the knowledge (i.e. strategies) which enables to select or configure a suited problem-solving method.

In this paper, we propose a *formal specification language*, the TASK formal language (TFL), which aims at dealing correctly with the control specification. The work is based on TASK, a methodology (Pierret-Golbreich, 1994) which is more convenient than other task oriented approaches, in particular KADS, for modelling flexible problem-solving and opportunistic reasoning. TFL aims at overcoming two main failures concerning existing knowledge engineering languages: *conceptual* modelling languages lack clarity and precision, whereas existing *formal* languages have limitations concerning the control specification for flexible problem-solving systems.

1.1 TFL requirements

TFL is a specification language whose goal is to comply with two main requirements: to be formal, that is to have a mathematically defined syntax and semantics; and to allow the formal specification of the control for flexible KBS.

- **Needs for formal language**
  A formal language is required to remedy the conceptual languages' lack of clarity and precision, in order to to enable the development of a KBS that behaves as it is intended to. It has been clearly recognised in the traditional software engineering that improving programming languages is not sufficient to improve the software quality. It is now clear that this quality depends mostly on the specifications of the system and its characteristics. This recognition gave rise to the study of formal methods in software engineering. At the same time in knowledge engineering, Newell's Knowledge Level hypothesis (Newell, 1982) emphasised the necessity of abstraction from particular representation languages. Newell suggested shifting the emphasis from representation to specification issues. Like any other system specification, the ME should be as precise, complete, unambiguous and consistent as possible. Unfortunately, the lack of clear semantics in the conceptual modelling language primitives leads to a lack of precision of the ME. A formal specification is required not only to ensure the system's correct behaviour, but also to enable a safe reusability of components. To be correctly *reused* during the conceptual design step of the ME, the "generic components" of libraries (e.g. Generic Tasks (Chandrasekaran, 1988), problem-types or canonical functions of CommonKADS (Breuker & Van de Velde, 1994) must be precisely specified. For instance, selecting the right inference requires certainty of its meaning. Unfortunately, these inferences have not got a precise semantics. For example, as Aben (1993) noted, the KADS inference "*abstract*" may have many different interpretations, thus when this component is reused to build a particular system, it is not guaranteed that it does what it was assumed to do. To analyse, validate, compare and reuse models of libraries or models of expertise, clear syntax and semantics of the language primitives are essential. Therefore, it is worthwhile studying how the formal specification languages traditionally used in software engineering could be applied to KBSs.

  The benefits of KBS formal specifications are multiple, since formal specifications allow us:

1. to better *describe* the functionality expected from the software system that is being designed: this is their first role. The KBS functional specification and its technical design and implementation are clearly separated. For instance, let "::" be the *application* operator and ";" the *sequence* operator. The semantics of the application of a sequence is given by the axiom $p; q :: \mathbf{D} = q :: (p :: \mathbf{D})$ (figure 9). This specification only focuses on the operator ";" functionality. It only expresses that $p; q$ applied to data D is equivalent to the application of q to the result of the application of p to D. This specification abstracts from the details of this operator's concrete implementation. In the same way, the axiom **resources(abstract) = abstract-model & ø** given in the *abstract* inference specification (figure 6) only expresses that a specific domain-model must be available to enable abstraction from patient observations to patient symptoms, but it abstracts from the concrete representation of this knowledge (e.g. rules, frames, logic etc.). The axiom **abstract** :: D = **apply-**

inference (abstract?) :: **D** (figure 6) expresses the connection between the inference *abstract* and the domain relation *abstract?*. It means that the result of the application of *abstract* is obtained in applying some domain knowledge, but the details of the inference mechanism operationalisation are left to the implementor.

2. to better *anticipate* the KBS behaviour, since a prototype can be derived from the specifications. A formal specification not only provides a description of the KBS expected functionalities, but also a means to forecast its behaviour. When dealing with algebraic specifications, as the TFL language does, different means may be used to derive such a prototype: rewriting, logic programming, code generation. In particular, in some cases, it is possible to transform the ADT axioms into an equivalent rewrite rules system (in using some Knuth & Bendix like algorithm) which enables one to obtain an executable prototype of the specified system. The notion of specification's "correctness" cannot be completely formalised. There is no formal way to prove that a specification is a "correct" description of the software to be designed. Indeed, to make a comparison between the formal specification and the system's expected properties, a precise description of the properties (that means *formal*) is required. But this is precisely the role of the formal specification, and in general only some informal "requirements" are available. Thus, running a prototype over "interesting" cases provides a possible way of validating the formal specifications and of testing that the KBS behaviour corresponds to that expected. ADT specifications have interesting properties for test selection (Gaudel, 1992; Marre, 1991b). Another way to check the adequacy of a specification is the use of the specification to prove consequent (expected) properties. Many software engineering works have already shown that algebraic languages and methods are interesting approaches for prototyping and proof. Indeed, the mathematical grounds of these techniques are quite well controlled, and there is a natural link between algebraic specification and rewriting.
3. to *prove* software correctness, since they provide a reference document that can be used. Specifications are considered by software engineering as a useful means to obtain programs which comply with given requirements. In this framework, software development is seen as resulting from a stepwise refinement process starting from a "high-level requirements specification" SP_0 to a "low level program" P. A suit of intermediate formal specifications $SP_1, ..., SP_n$, is produced during this process, which is called " implementation". The program P is said to be an implementation of each specification SP_j (Orejas et al., 1992) and in this context, SP_i is considered to be also an implementation SP_j, whenever $i > j$. Different interpretations of "implementation" are available in this framework of software development.

   Our specification language benefits from a "loose" semantics which allows one to specify only the desired properties, at each level. In our language, SP' is an implementation of a specification SP if it "specialises" SP in adding new property definitions to it. Thus SP' restricts the set of possible models of SP. Nevertheless, each model of SP' must still remain a model of SP. In that way, all the requirements defined in SP_i ($0 \leqslant i < n$) are satisfied by SP_n. Existing ADT techniques allow one to check the correctness of the successive intermediary implementations from SP_i to SP_{i+1}. The relation between P and SP_n is different since P is not a specification but a program. The program P describes the same object as SP_n but in a "concrete" operational language. The results obtained with P should be deduced from the specification SP_n. A method to verify it consists in deriving a set of tests from the specification P, and to watch at their satisfaction by the program P. There exists some tools (e.g. LOFT developed in the LRI by Marre (1991a) to verify that P really satisfies all the requirements specified in SP_n.
4. to decide on the software *reusability* in exploiting these reference documents. Formal specifications can serve as reference documents to decide if a given software module can be reused for another purpose (different domain, application, task, etc.) The abstract and precise nature of formal specifications ensures the reuse of components of libraries with a more safe interpretation. For example, the ambiguity of the KADS inferences due to their informal description makes it difficult to select or adapt them for a particular application. On the opposite, for example, the formalisation in first order logic of the inferences proposed by Aben (1995) makes their

assumptions and properties more explicit. In a similar way, the TFL specification of the inferences or tasks given by ADT axioms provide them with a precise semantics. For instance, an axiom defined in an ADT **Hierarchy**: classify (i, C) < = > i is-a C and non (i is-a C' and C' < C) clearly defines the semantics of the *classify* action: "to classify an object i within a hierarchy means to find the most specific class C this instance i belongs to".

- **Necessity to specify a dynamic control**
  Proposing a new formal language is required to overcome existing limitations concerning flexible problem solving and control. Indeed, non-deterministic applications is the most favoured field of KBSs. The most important difference between traditional systems and KBSs is their declarativity and internal[1] non-determinism. Most often in a KBS, the way to solve a problem is not known in advance. The problem-solving process cannot be described by a fixed procedure. Thus, it cannot be predefined, but has to be dynamically computed during the resolution, in using some declarative knowledge. This internal non-determinism precisely gives rise to a *control problem*: "which is the best potential action to be executed?" (Hayes-Roth, 1985). Most of the formal languages which have been recently proposed (e.g. $(ML)^2$ (Van Harmelen & Balder, 1992), K_{BS} SF (Jonker & Spee, 1992), FORKADS (Wetter & Schmidt, 1991), KARL (Fensel et al., 1991), QIL (Aitken et al., 1992) are based on the KADS methodology. KADS doesn't really provide satisfactory answers to the control modelling. First, in KADS dynamic knowledge is described in a procedural form (by a task), and a task is associated with a single problem-solving method task body with a fixed control structure; second, the definition of strategic knowledge is not very precise in KADS I and the strategy layer is removed from the ME (to the meta level) in KADS II; third, the KBS control process is not considered since the KADS ME aims at specifying the application expertise and not the reasoning control. Thus, the existing languages are generally more suited to KBSs with fixed tasks decomposition.

1.2 Dynamics semantics

This section does not aim at presenting an exhaustive review of the formal languages found in knowledge engineering (refer, for example, to Fensel and van Harmelen (1994), Treur and Wetter (1993) or van Harmelen and Fensel (1995)). The goal is only to provide a short analysis of how these approaches specify the *dynamics* of KBS. The dynamic behaviour depends on several components:

- the object system problem-solving knowledge and strategical knowledge,
- the meta-system's *own* problem-solving knowledge and strategical knowledge.

The existing languages are now surveyed, emphasising: three dimensions: (i) how they specify the object-system problem-solving knowledge; (ii) do they enable flexible problem-solving thanks to strategical knowledge; (iii) do they specify the meta-system processes involved in controlling the resolution, in particular for dynamic choices. Here are the most famous languages, from the most procedural to the most declarative:

- K_{BS}SF is based on algebraic specifications. It is used to describe three layers of objects: data, knowledge, behaviour. Data are specified by ADTs, knowledge is specified in using order sorted logic. Inference roles are specified by parametrised signature types, while their action is specified by a Bmodule. A Bmodule consists of Input/Output parameters, local variables and a body which specifies a procedure to achieve the required behaviour. A task is specified in the same way,

[1]Opposed to external non-determinism, which means that a program can deliver several outputs for the same input.

by a set of Bmodules. The behaviour description is given in terms of Bmodules, in using "a control oriented specification language" which offers control statements like assignment, task call, choice and iteration, and also operations on theories.

- In KARL the problem-solving process of a KBS is modelled at the task layer. KARL uses Procedural-KARL, a variant of dynamic logic (Harel, 1984) but restricted to regular and deterministic programs. Procedural-KARL "can be used similar to a procedural language" (Fensel & van Harmelen, 1994) to describe the control flow between the different inferences of an inference structure and tasks are "similar to procedures in programming languages". In Procedural-KARL, the primitive programs correspond to inferences calls. Programs are combined by sequence, loop and alternative into more complex programs which correspond to tasks. Thus, KARL allows the declarative description of a fixed control flow, but not of dynamic control. Moreover, at the inference layer, there is no non-deterministic choice of instantiation of the inference actions, since KARL uses the complete minimal Herbrand model for the inference execution. These restrictions have been voluntarily made to provide executability: "the logic language is restricted to Horn logic and the specification of control requires a deterministic (over)-specification (Fensel, 1995a)". On the one hand, KARL only aims at specifying the object-level reasoning process and is not concerned by the meta-level reasoning. On the other hand, consequent to later restrictions, KARL is not concerned with the specification of control for non-deterministic choice of knowledge. In return, KARL does not offer flexibility.
- $(ML)^2$ uses Quantified Dynamic Logic (QDL) (Harel, 1984), a multi-modal logic, to represent the KADS task layer. Tasks are viewed as QDL *programs*, which are complex QDL expressions. The primitive programs correspond to the inferences. Inferences are specified by a predicate together with the test operator—? They use a *state* variable to store the input/output pairs computed at each step. Two primitive program statements are available: assignment and test. Different constructors are provided to combine programs: sequence, non-deterministic iteration and choice. The non-deterministic operators enable a declarative description of dynamic and also fixed control. These operators and the state variables make $(ML)^2$ more flexible than KARL. The $(ML)^2$ non-deterministic operators offer broad potentiality. They might enable one to specify a non-deterministic choice between several problem-solving methods for a task, or conditions stating which inference should be selected. However, non-determinism has been limited to the inference layer, and this potentiality is not really illustrated on tasks. Even though a description of non-deterministic control flow should be possible, $(ML)^2$ has been used only for a fixed task-decomposition, the usual content of the KADS task layer. Like KARL, $(ML)^2$ uses a procedural representation of control, and does not aim at representing strategic reasoning. Thus no specification of strategical knowledge is provided. Both languages focus on specifying the KBS reasoning but not the control over this reasoning. The meta-system control is not specified and implicitly relies on an interpreter/theorem prover.
- QIL is based on a multi-modal logic. QIL is more flexible since it does not represent problem-solving knowledge by fixed task decomposition. Tasks plans are dynamically generated by planning rules. The temporal modality is used to specify the planning process. General rules, which specify how to form a plan that meets a goal, are represented as beliefs of an agent about the future. However, whereas such rules can explicitely express which action should be selected, nothing is provided to choose one action from among several possible ones. Thus, no way of specifying strategical knowledge is provided.
- In contrast to the previous languages, which are all based on KADS, DESIRE is dedicated to *compositional* architectures. DESIRE uses temporal logic. The state changes and temporal aspects are explicitely covered by the formal semantics of DESIRE. A current state in time is represented by a partial model. The reasoning process is modelled as a function between such partial models. Whereas in KADS hierarchical decomposition is only available at the task layer, in DESIRE the control structure is not separated at a distinct level, but is included in each composed component itself. DESIRE makes distinction between meta-level and object-level, which enables one to specify the flexible control of inferences. The meta-level describes the

dynamic aspects of the object-level in a declarative fashion. The decomposition into several modules, each containing its own object-level knowledge and meta-level knowledge, allow one to specify any number of levels to describe the complete system. The global control is represented by a rule formalism at a separate level. By this means, DESIRE permits us to express dynamic control decisions. But the well known disadvantage of this rule formalism is to make the global behaviour difficult to predict. The execution of specification is obtained by an interpreter.

Although its representation of the global control has a flavour of the symbol level, DESIRE provides a means of representing strategic reasoning. In conclusion, DESIRE is the single existing language that offers sme capacity of specifying strategic knowledge and flexible problem-solving. However, because DESIRE is not very intuitive, its use may be limited to some specialists.

The aim of TASK is to provide a task-oriented framework that does not elude the difficulty due to the specificity of KBSs compared to traditional software, but takes care of the complexity related to their non-determinism. A specification language for KBS must enable both procedural *and* dynamic control. This point is basic in the TFL language. It has implied the choice of a unique specification formalism because of the dual role of the problem-solving knowledge. Indeed, problem-solving knowledge is viewed as operative knowledge when applied to data, and as *data* when the control operates on it. Thus, a major contribution of the approach is the use of a single knowledge specification formalism: Algebraic Data Types (ADT), which are used to represent data and knowledge in an implementation independent way with a precise syntax and semantics. For each piece of problem-solving knowledge, a module expresses both its static and dynamic semantics.

Section 2 describes the modelling primitives according to the TASK methodology. In particular, the notion of process is clarified, and a reflexive hybrid control combining the opportunistic and hierarchical approaches is presented. In section 3 the mathematical object of *process-module*, which allows an abstract and modular description of the problem-solving knowledge, is introduced, and the whole specification of the dynamic behaviour of a KBS is given.

2 The TASK model

This work is devoted to a formal specification of the expertise models according to the TASK methodology (Pierret-Golbreich, 1994).

2.1 The TASK methodology

The task methodology is based on the following principles:

- **Functional view of the problem-solving knowledge:** expertise is modelled as several tasks (agents) which cooperate to solve a global problem. Each task has a partial view of the problem solving, and is assigned a specific problem-solving competence (a typical function).
- **Categorisation of knowledge:** expertise is divided into three types: domain knowledge, problem-solving knowledge, control knowledge.
- **Interaction hypothesis:** task oriented modelling is based on the hypothesis that these three categories of knowledge are inter-related. This is a similar hypothesis to the Interaction Hypothesis of Generic Tasks (Chandrasekaran, 1988). A new concept, the Task-Module, has therefore been introduced to refer to the whole bundle of knowledge concerning an expertise module, whatever its level of genericity. A Task-Module is a pair [competence, knowledge]. The competence part gives an abstract specification of the module competence. The knowledge part is composed of three elements:

  resources refers to the domain modules required by the task.

  processes refers to the operative modules the task can use to reach its goal.

  strategies refers to the strategic modules the task has to choose for the pertinent problem-solving module in a current context.

- **Modularisation:** each type of knowledge is organised into several modules. Domain knowledge is structured into domain *structures*, dependency subnets corresponding to the different domain relations. Problem-solving methods are structured into *processes*, while strategic knowledge is organised into *strategies*. Specialists of a problem type are organised into Task-Modules.
- **Hybrid control:** the model of control combines opportunistic (as defined in BB1 (Hayes-Roth, 1985) and hierarchical approaches of control (as defined in CRYSALIS (Terry, 1983) or KADS (Wielinga et al., 1992).

2.2 The TASK primitives

Among different possible control approaches, a sharing data communication model[2] has been chosen. The model control is grounded on two main primitives, which are *data* and *process*. *Data* models the working memory which contains all the current information on the case, the active strategies and foci. *Processes* model the problem-solving knowledge which operate on the data. The problem-solving activity consists in applying processes to data until the problem-solving goal is met.

2.2.1 Domain knowledge

The modelling of the domain offers a clear distinction between two kinds of entities: *domain knowledge* and *information*:

- **Domain knowledge** is the static knowledge representing the "universe" of an application. The modelling primitives look like the epistemological primitives of KL-ONE (Brachman & Schmolze, 1985). They are very similar to that proposed in CommonKADS (Wielinga et al., 1992, 1993) in terms of **concepts, properties, values, relations** and particularly **domain structures**. Each *domain structure* contains the knowledge about a particular relation. It supplies a particular *view* on the domain according to a precise type of relation. For instance, a causal structure collects all the statements about the domain objects which are linked by a *cause-effect* relation. They form a partition of the knowledge base according to these relations.
- **Information** refers to the factual data and assertions about the case: they model the current state of the data concerning the case. A piece of information describes the value of a concept property for an instance. For example, information about the current case could be: *The temperature of the patient 'Jean is 37°, he hasn't got fever, the stress of the patient 'Lug is high...*

  The notion of information is crucial in the model. Indeed, the initial case is defined by a set of information stored in the working memory. Problem solving consists in **completing the set of known information** by applying the domain and problem-solving knowledge, according to the strategical knowledge.

2.2.2 Problem-solving knowledge

The problem-solving knowledge is the operative knowledge of an application. Each type of problem-solving knowledge is modelled by a **process**. A process describes an inference process (whatever its type) that could potentially operate on the data. So, it can modify the state of the data. However, this notion is not similar to that of a process in software engineering. The latter is only operative, and thus only defined, by its action on data. On the contrary, in knowledge engineering a process can additionally be viewed as a **data** with respect to the control activity which has to choose one process to be applied among several potential processes. Thus the semantics of a process is not only concerned with the result of its application to the data. Indeed, the control also needs to access the process's features in order to evaluate their relevance in the resolution, with the purpose of applying **only** the most interesting one. Therefore, a process is defined both by its semantics as *action* (dynamic semantics) and by its semantics as *data* (static semantics):

[2] A blackboard model.

Expression of dynamic semantics (application of a process to the data)
Three categories of processes are distinguished according to their means of application to the data: inferences, composed process, task-modules.

- *Inferences*
  A process is built from building blocks, the inferences. *Inferences* represent the atomic actions that a system performs to change the problem-solving state. Applying an inference to data corresponds to performing implicit operations tied to a domain structure. The action corresponding to a relation R of a domain structure consists in infering, from all the occurrences of R in the domain structure, the information b when the piece of information a is present in the data.[3]
- *Composed processes*
  A case is solved by the application of several inferences to the data. *Composed processes* are pseudo "methods" depicting links between less complex processes. They are described as pseudo "programs" with the following constructors:
  — ; — represents the sequence,
  — * the non-deterministic loop,
  — ? the test,
  — ⊕ — the non-deterministic choice.
  In a process, the non-deterministic choice operator expresses that several processes are available to realise the same action in different ways. Two categories of composed processes are distinguished:

  1. *deterministic methods* correspond to static resolution plans whose decomposition is already determined before the running. They are represented as combinations of processes which do not involve the non-deterministic operator.
  2. *non-deterministic methods* correspond to dynamic resolution plans. The process decomposition is now known *a priori*, before its application to data, but dynamically generated during the resolution thanks to the control knowledge. They are represented with the use of the non-deterministic operator.

- *Tasks-Modules*
  Task-Modules are "specialists" of a problem type. They capture in a unique entity the whole knowledge implied in an expert module. A task is composed of:
  —*a competence*, describing the problem-solving situations the task is concerned with.
  —*resources*, set of local domain structures required by the task.
  —*processes*, set of local processes that could be used to perform the task.
  —*strategies*, set of local strategies used by the task to dynamically build a resolution plan suited to a particular case.
  A task local control uses the task local strategies to get a method to achieve the task: if the task local processes consist of a non-deterministic process, then the method is obtained by selecting one process from among several possible ones thanks to the strategies; if they consist of a set of sub-processes, then the method is dynamically generated by combining these sub-processes (e.g. set of sub-tasks) according to the strategies. This allows an important cut in the search space. The result of applying a task to data is the same as the result of applying the process dynamically generated thanks to the local task control.

  For example, **classification** is a typical task that may be realized by different processes for instance by heuristic classification, hierarchical classification, etc. A local strategy of the Classification Task expresses, for instance, that the process choice might be based on the nature of the domain structure available. It should be noted that opposed to KARL, a process can describe non-deterministic programs.

[3]The implicit inference action related to a domain structure is not always so simple, but could be formulated in a similar way from a set of domain structure relations.

Expression of static semantics (process intrinsic features)
In our modelling framework, a process is not only viewed as an action but also as a static object with its own features. These features are used by the control processes to make selection and choice when needed. A process is therefore characterized by the following explicit features:

1. a *name* refers to the process.
2. *input* and *output* are the set of concepts the process deals with.
3. *resources* are the required domain structures.
4. a *pre-condition* is the necessary condition the current input objects must satisfy so that the process can be activated.
5. a *post-condition* is the necessary condition the output objects have to satisfy after the process has been applied to the data.
6. *(sub) processes* and *(sub) tasks* used in the process body.

This approach presents two important advantages, related to knowledge acquisition and to the control knowledge definition:

- The first advantage is that the characteristics of a composed process are, by definition, generated from the characteristics of its components. In that way, when the characteristics of all the inferences are known, the characteristics of **all** the composed processes are **automatically derived**. Therefore, only the characteristics of the inferences have to be defined during the acquisition process.
- The second advantage is the use of a unique structure for the processes features, the task's competence and the foci (cf. section 2.2.3). In that way, the control function is viewed as a function working on a unique data structure which aims at matching characteristics of the same type.

2.2.3 Strategic knowledge
To solve the control problem requires that the working memory not only contains the information about the case, but also the strategic knowledge that enables the control to compute a choice from among applicable processes. *Strategies* which fulfil this purpose are built from building blocks called *foci*:

a **focus** corresponds to an heuristic which privileges some subset of processes. A focus is expressed as a constraint on the problem-solving knowledge. According to the type of these constraints, a focus represents a *data* driven control—which privileges knowledge operating on some data (within a medical application, for instance, "*prefer actions on symptoms of diseases*")—a *goal* driven control—which privileges knowledge which could help to reach a given resolution state (for instance, "*prefer actions which conclude on cardio-vascular diseases*")—or an *action* driven control—which privileges knowledge directly dealing with specific actions (for instance, "*prefer classification processes*"). A focus can express either generic preferences like "*actions on hierarchically structured data*" or specific preferences like "*actions on pneumonia*": control is achieved at several granularity levels. A focus is taken into account by the control only when it is put in the working memory, and then said to be *activated.*

a **strategy** is responsible for activating and deactivating the foci or for expressing their combination. Indeed, a heuristic is generally not valid during the whole resolution. A strategy activates or deactivates sub-strategies and foci. In this way, any number of control levels can be defined. Each level manages its sub-levels. A strategy has a *name* and a *body* which is an expression in a dedicated language. The basic terms of this language are:

- "ε" represents the finishing action. All strategies have to end with this operator.
- "**post-strategies**" represents the action of activating a set of strategies on the working memory.
- "**post-foci**" represents the action of activating a set of foci on the working memory.
- "**stop-strategies**" represents the action of deactivating a set of strategies.
- "**stop-foci**" represents the action of deactivating a set of foci.

The operators are:

- the sequence: ";"
- the condition: "(if— —)"
- the loop: "(loop— —)"

This recursive definition of the strategic knowledge, inspired from Albert et al. (1992) allows the specification of multiple meta-levels.

2.3 *The control model*

The model is reflective: the control is specified in terms of processes similar to the problem-solving ones, thus a uniform description of all the dynamic knowledge is possible. These control processes are invoked only when a choice between alternative processes is needed. Such a case occurs either for the application of a composed process, including the non-deterministic operator, or for the application of a task. Since it is only used when it is really necessary, the control is more efficient.

The schema of the global model of control is given in Figure 1. Control usually drives the resolution. An opposite approach is used here: problem solving calls for the control. Problem solving is completely described in terms of processes application.

1. The application of an inference consists in the addition, suppression or modification of the current information according to the domain knowledge tied to the inference.
2. The application of a deterministic composed process corresponds to the application of the sub-processes it is made of.
3. The application of a non-deterministic process $p \oplus$ to[4] the data D is replaced by the application to D of a choice process operating on p and q.[5] The result of applying $p \oplus q$ to D is thus equal to the result obtained in applying the "chosen" process to D. The intervention of the control thus brings back to the previous cases 1 or 2.
4. The application of a task is also brought back to case 1 or 2 through the control. The control exploits the knowledge embedded in the task in order to get a deterministic process or an inference which satisfies the current data (information and foci) and its competence. Performing the task is then equivalent to applying this process to the data. Thanks to the Task-modules, the resolution is more efficient since they make a local control possible: local strategies activate

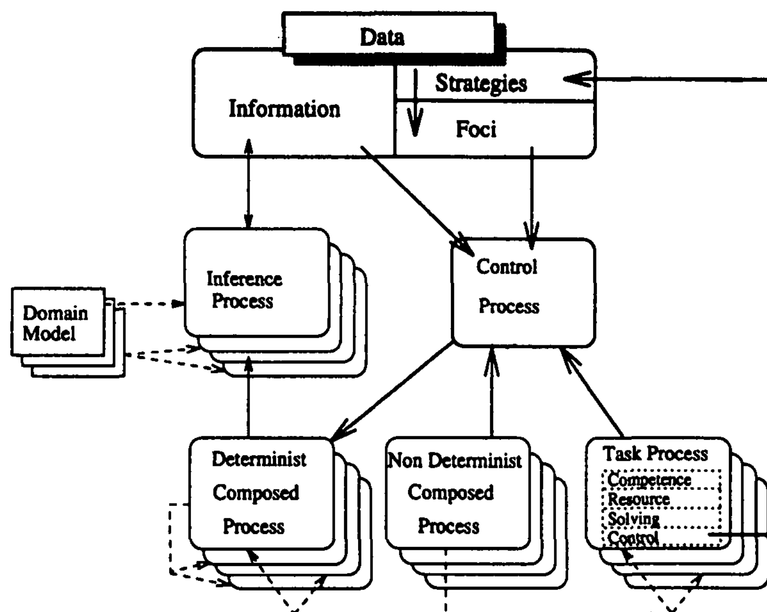

Figure 1 The global model of control

(deactivate) heuristics which are specific to the task, and the search space is reduced to the local processes.

As problem solving is viewed in terms of processes application to data, before the problem-solving process starts, initial information describing the case to be solved must be set up and an initial process must have been defined. The initial process models the main task the system has to perform. This process may be of different nature, depending on the available problem-solving expertise. If a fixed chaining of sub-processes is known *a priori*, then it is a deterministic process. In other cases, it is described by a main task whose

- competence expresses the main problem-solving situation that is, all that is known about the problem to be solved.
- processes expresses the sub-processes or sub-tasks the task can make use of to meet its goal.
- strategies express the available heuristics to reach the task goal or the pertinent strategies to build a problem-solving method.

The problem solving activity starts by applying this process to the initial data and is then pursued according to the general scheme of processes application described above.

3 TFL formal specification of KBS dynamic behaviour

The different categories of knowledge are specified in a single formalism: Algebraic Data Types (ADT). This choice of ADTs as the single formalism used to specify both static *and* dynamic knowledge is an important specificity of the approach. It is particularly convenient to reflect the double nature of processes: dynamic, from the resolution point of view, but static from the control point of view. Indeed, when applied to data, a process has an active role; when the control processes of the meta-system are applied to them, then a process is viewed as a static data.

An ADT presentation (or more simply a specification) is a triple ($\mathcal{S}$, Σ, Ax) where ($\mathcal{S}$, Σ) is a signature and Ax is a set of first order logic formulae (but generally limited to positive conditional equations). A signature $\mathcal{S}$, Σ), is a finite set $\mathcal{S}$ of sorts (i.e., type-names) and a finite set Σ of operation-names with an arity (defining the sorts of their domain and codomain). The syntax which is adopted is that of PLUSS (Gaudel, 1984, 1985), a specification language developed in the framework of the ASSPEGIQUE project (Capy, 1987). The semantics associated with a specification in PLUSS is of the "set of models" type (opposed to the initial approach where there is a unique—up to isomorphism—algebra validating the axioms). Several PLUSS constructors are used:

- Each ordinary ADT is introduced by the keyword **spec** followed by the module name.
- ADTs which are required are introduced by the keyword **use**.
- A new name can be given to a part or to a whole specification by the primitive **renaming ... into ....**
- A generic specification is introduced by the keyword **generic spec**. A generic specification is parameterised by formal parameters. A formal parameter specifies the minimal properties to be satisfied by the effective specifications that can be substituted to it in the instantiation process. A generic specification is similar to an ordinary specification, but all the objects in the signature are not declared.

For example, in the specification **LNAT** (Figure 2), the set of values of the sort LNat are the sorted lists of integers. The operations ϕ (the empty list), *cons* (which adds a new integer to a list), *head* (the first element of a list), *tail* (the queue of a list) and the predicate *sorted* are the operations

[4]Any process that has an operator $\oplus$ can be considered as a non-deterministic choice between several totally deterministic processes.
[5]Only the processes matching the active foci are selected and the most appropriate one is chosen from among them.

```
spec LNAT as
use NAT, BOOLEAN
sorts LNat
operations
  ∅ : ↦ LNat
  cons - - : Nat × LNat ↦ LNat
  tail - : LNat ↦ LNat
  head - : LNat ↦ Nat
  sorted - : LNat ↦ Boolean
axioms
  head( cons( N, L)) = N
  tail( cons( N, L)) = L
  sorted( ∅) = true
  sorted( cons (N, ∅)) = true
  sorted( cons (N, cons( M, L))) =
       and (≤ (N, M), sorted( cons( M, L)))
where N,M : Nat
  L : LNat
```

Figure 2 A specification of sorted lists of integers

allowed on these values. The axioms only express *what* is a sorted list of integers and not *how* such a list should be implemented. This ADT uses two predefined ADTs, **NAT** and **BOOLEAN**. The semantics associated with the specification **LNAT** is the class of finitely generated algebras which validate its axioms.

By definition, ADTs have been developed to specify data types. Their favourite field of application is that of procedural programs. Therefore, it was necessary to adapt classical ADTs, in order to enable the specification of processes and strategic reasoning. The formalism presented here has been inspired by work on algebraic specifications for concurrent processes (Kaplan, 1987). The essential ideas that influenced it are the definition of *process specifications* which are used to simplify the development of specifications and an *application* operator through which processes operate on data (dashes represents the place of the arguments in the signature of the operator):

$$- :: - : \textit{process} \times \textit{data} \rightarrow \textit{data}$$

This section is now focused on KBS specification done by a knowledge engineer and thus presents the main concepts he has to know for that purpose. The novel tool of *process module* is particularly outlined in section 3.2. Its syntax is first given in section 3.2.1. The semantics which is assigned to the set of all the process modules of an application is given in section 3.2.2.

3.1 Data and information

The working memory contains all the current information about the case, the set of active strategies and foci. It is specified by the module **Data** (Figure 3)[6] which requires the different specification modules, Information (Figure 4), Strategy and Focus (cf. Appendix).

3.1.1 Data

The working memory **Data** specification is represented as a record whose fields are associated respectively with the set of information about the case (the current state of the domain data), the set of active strategies and the set of active foci. The class of models which validate this specification is composed of all the possible states of the working memory. Two particular ADT (see Pierret-Golbreich and Talon, 1994 for details) are used for that purpose, **Set** and **Record-of**:

[6]The first letter of a Sort is a lower character whereas a Specification module has an upper character.

```
spec Data as
  Record_of(        < STATE, Set-Information >,
                    < STRATEGIES, Set-Strategy >, < FOCI, Set-Focus >)
  renaming Record into data
```

Figure 3 The working memory specification

```
spec Information
  use Property, Instances-concepts, Value
  sorts information
  operations
      — in — is — : property × instances-concepts × value ↦ information
```

Figure 4 Signature of information

- **Set** is a generic algebraic data type which specifies sets of items. The constant ϕ stands for the empty set. The operation **&** stands for the addition of an item to a set **and** the union of two sets. Assuming that the integers are represented by the sort **nat**, the set of integers [2, 3, 4] is represented by **2 & 3 & 4 &** ϕ in **set-nat.**[7] **Set-Information, Set-Strategy** and **Set-Focus** are defined as instantiations of this generic ADT **Set**.
- **Record-of** is an ADT defined in Kaplan (1987). The operator "— [*label*\—]" provides a primitive for modifying records field by field and the operator "—[*label*]" provides an access to a particular field (the first argument of both operator is a record). So the assertion "E = D[STATE\$I2$& D[STATE] & ϕ]" expresses that E is a record equal to D except for the field STATE which contains an additional value $I2$.

3.1.2 Information

The operator—**in**—**is**—is a generator of the sort **information** (Figure 4).

The sorts **property, instances-concepts** and **values** are respectively modelling the properties, the domain objects and the range of values of those properties. The corresponding algebraic data types are given in (Pierret-Golbreich and Talon, 1994). Information could be "**age in Jean is 43**" and current state of domain data could be "**age in Jean is 43 & pressure in Jean is 100 &** ϕ".

3.1.3 Boolean expressions

The choice of one problem-solving process rather than another one depends on the current state of the resolution. The specification of the problem-solving knowledge thus needs to be able to express conditions relative to the current information. Such a condition cannot be defined in a fixed way and involves the notion of **dynamic** condition. The solution proposed by Kaplan (1987) for modelling such dynamic conditions consists in defining a new sort **exp-bool** denoting a range of expressions which are applied on the current domain data (**set-information**) through the application operator <::>:

$$- <::> - : exp - bool \times set - information \rightarrow bool$$

[7]The name of sorts prefixed by **set-** are used to give a name to the corresponding set sort, Specification sets are prefixed by **Set-**).

This operator plays a role similar to the process application operator. All the boolean expressions are defined in **exp-bool** by axioms like: $exp <::> D = E$ where D is a term of **Set-Information** and E is a boolean term on **Set-Information**.

A boolean expression is transformed into a process by the test operator—? which takes a boolean expression as argument. The application of such a process succeeds only if the dynamic application of the boolean expression to the same data provides the value true.

3.2 Process and process modules

The problem-solving knowledge is specified with the help of a specific mathematical object *process module* which allows us to describe the reasoning knowledge in an abstract and **modular** way. Unlike a process specification in Kaplan (1987) which has always got a semantics given by an associated classic specification *SEM(P)*, a process module has no semantics. A process module is a "*syntactic sugar*" which simplifies the specification of a piece of problem-solving knowledge. It is the set of *all* the process modules which has got a semantics. For that purpose, these processes are associated with a classical specification **Process** which covers the given specification of the process modules and all the "machinery" needed for the writing of the problem-solving knowledge. By definition, the processes semantics is the classical semantics of the specification **Process**, that is the class of its models. The KE is not concerned with that specification: he only has to specify the application processes with the mathematical primitive of the process module. The associated specification **Process** is automatically built frol all of theses process modules. The Process semantics is shortly explained in section 3.2.2.

3.2.1 Syntax of process modules

The syntax of a process module is given in Figure 5. The different fields are optional. Generally, a process module is used to specify one *module* of knowledge of an application, thus all the fields are not necessarily used in each process module, but only the ones needed. Two sets of fields are distinguished: fields for problem-solving knowledge, and fields for strategic knowledge. A diagnosis task concerning a medical application using the heuristic classification method illustrates the TFL specification of the problem-solving and strategic knowledge.[8]

Fields for the problem-solving knowledge (Figure 5)

Any type of processes, inference, composed process and task is specified by use of process modules. The relevant fields (shown in Figure 5) are used for each process type:

- **Inferences**

  The name of an inference is stated in the field **inferences**. It enables to generate the declaration of the corresponding atomic process in the signature of the specification module Process, through its required spec Process-Name (Figure 9).

  The specification of the inference characteristics is given by the equations of the field **axioms for inference characteristics**. Predefined operations (**input, output, resources, pre, post, process** and **tasks**) are declared for each inference characteristic in the signature of the spec Process (Figure 10). The axioms given in this field enable us to describe the specific values of the characteristics of the inference. The definition of the pre- and post-conditions involves the use of the boolean expression application operator $<::>$.

  The specification of the inference dynamics is expressed by a single axiom in the field **axioms for inference application**. This axiom form is of the type: $I :: D =$ **apply-inference**$(R) :: D$, in which I

[8] In this example, the specification is a Knowledge Level specification of the KBS behaviour, but placed at the application level of abstraction. The domain knowledge and the meta-knowledge specification, which is of the utmost importance for Knowledge Acquisition, has not been addressed here (see Pierret-Golbreich & Talon, 1994).

```
process module module name
        inferences I
                inference names
        axioms for inference characteristics :
                AX^inferences_characteristics : definition of the inferences characteristics
        axioms for inference application :
                AX^inferences_application : definition of the inference application to data
        composed process C
                composed process names
        axioms for composed process application
                AX^composed process_application : definition of the composed process application to data
        tasks T
                task names
        axioms for tasks
                AX^task_definition definition of tasks

        foci F
                focus names
        axioms for foci
                AX^foci definition of foci
        strategies S
                strategy names
        axioms for strategies:
                AX^strategy_body definition of strategy bodies
```

Figure 5 The syntax of process modules

is the name of an inference and R is the name of a relation. The operation **apply-inference** expresses the connection between an inference and the associated relation of a domain structure.

For example, consider the inference **abstract** of a medical diagnosis application, which derives some symptoms from patient observations. Figure 6 shows the process module corresponding to this **abstract** inference. The patient is an object of the sort **patient**. The inference applies the domain knowledge specified in the domain structure **abstract-model** linked to the relation **abstract?** in order to derive symptoms from observations. The equations in Figure 6 mean that:

— the name of the inference is abstract.

— the static semantics of the inference, viewed as data, is given in the field **axioms for inference characteristics**:

1. the input and output of *abstract* must be instances of the concept patient.
2. **pre(abstract)** represents the pre-condition of *abstract*. It is expressed by a boolean expression. The result of its application by the operator <::> to the environment **D** is true if there is an information **I1** in **D** such that **abstract?** ($I1$,$I2$) is satisfied. The post-condition of *abstract* is given in the same way, but information $I2$ must have been added to the current state of data $D[STATE]$.
3. only one domain-model is required as a resource by *abstract*: the *abstract-model.*
4. *abstract* owns a single process, *abstract* itself, and no sub-tasks.

— the dynamic semantics of the inference, viewed as a process, is given in the field **axioms for inference application**: only one equation is needed, which states that the result of the inference application to the data **D** is provided by the domain knowledge linked to the relation **abstract?** applied to these data.

```
process module Abstract
  inferences abstract
  axioms for inference characteristics
      input(abstract) = c-patient & ∅
      output(abstract) = c-patient & ∅
      pre(abstract)<::> D = I1 ∈ D[STATE] ∧ abstract?(I1,I2)
      post(abstract)<::> D = I1 ∈ D[STATE] ∧ abstract?(I1,I2) ∧ I2 ∈ D[STATE]
      resources(abstract) = abstract-model & ∅
      processes(abstract) = abstract & ∅
      tasks(abstract) = ∅
  axioms for inference application
      abstract :: D = apply-inference(abstract?) :: D
  where
      I1, I2 : information
```

Figure 6 Specification of the **abstract** inference

- **Composed process**

  A composed process is specified in the same way. The equations of Figure 7 mean that:
  - —The name of the process is heuristic-classification.
  - —It should be noted that the static semantics of a composed process require no specific axioms since it is *automatically derived* from the characteristics of its components together with the semantics of the composition operators. For example, the semantics of the operator ";" implies that the resources of *heuristic-classification* is the set of models obtained by union of the resources of its sub-processes: abstract-model, heuristic-model, hierarchic-model.
  - —The dynamic semantics of a composed process describes the result of its **application** to data, and is specified by a unique axiom which expresses the equality between the name of the process and its decomposition into sub-processes. The heuristic-classification method defined in Figure 7 is a fixed plan which applies in sequence the inferences abstract, match and refine.
- **Tasks**

  The name of a task is stated in the field **tasks**. It will be used to declare the tasks involved in the signature of the specification module Process.

  The specification of the task characteristics is given by the equations of the field **axoims for tasks**.

  The names of the local strategies of the task are stated in the field **strategies**.

  The specification of the task strategies are given in the field **axioms for strategies**. They are used in order to generate the axioms of the operator strategies declared in the **spec** Task (Appendix Figure 15.

Figure 8 shows the process module describing the heuristic classification task. This task exhibits the competence of heuristic classification, operates on patients and diseases, and requires three resources. Since it is not efficient to apply all three inferences if only some of them are sufficient, a task local strategy is defined so as to apply only the useful inferences. Thus, the task problem-solving knowledge is described by the set of processes {heuristic classification, abstract, match,

```
process module Heuristic-Classification
  composed process heuristic-classification
  axioms for composed process application
      heuristic-classification = abstract; match; refine
```

Figure 7 Specification of the **heuristic classification** method

```
process module T-Heuristic-Classification
  tasks t-heuristic-classification
  axioms for tasks
      input(t-heuristic-classification) = c-patient & c-disease & ∅
      output(t-heuristic-classification) = c-disease & ∅
      pre(t-heuristic-classification)<::> D = ∅
      post(t-heuristic-classification)<::> D = I1 ∈ D[STATE] ∧ bottom?(I1,I2) ∧ I2 ∈ D[STATE]
      resources(t-heuristic-classification) = abstract-model & heuristic-model & hierarchic-model & ∅
      processes(t-heuristic-classification) = heuristic-classification & abstract & match & refine & ∅
      tasks(t-heuristic-classification) = t-heuristic-classification & ∅
      strategies(t-heuristic-classification) = s-heuristic-classification

  foci f-patient, f-hc, f-hcl, f-hc2, f-hc3
  axioms for foci
      input(f-patient) = c-patient & ∅
      output(f-patient) = c-patient & ∅
      tasks(f-hc) = t-heuristic-classification & ∅
      processes(f-hcl) = heuristic-classification & ∅
      processes(f-hc2) = match & ∅
      processes(f-hc3) = refine & ∅

  strategies s-heuristic-classification, s-hcl, s-hc2
  axioms for strategies
      body(s-heuristic-classification) = post-strategies(s-hcl & s-hc2); ε
      body(s-hcl) = (if ¬ ( (P in _ is _ ) ∈ D[STATE] ∧ is-symptom(P))
                     post-foci(f-hcl & ∅) ); ε
      body(s-hc2) = (if ( (P in _ is _ ) ∈ D[STATE] ∧ is-symptom(P) )
                     post-foci(f-hc2 & f-hc3 & ∅) ); ε
```

Figure 8 Specification of the **t-heuristic-classification** task

refine}, and the strategy **s-heuristic-classification** enables the dynamic configuration of a method with a relevant control structure, since it prescribes which process to select and when.[9] It expresses that if no symptoms have been deduced, the heuristic-classification process is privileged; if not, processes match and refine are privileged.

Fields for the strategic knowledge (Figure 5)

The strategic knowledge is specified by use of the fields foci and strategies.

- **Foci**

  The process module shown in Figure 8 contains the specification of several foci. Foci **f-hcl**, **f-hc2**, **f-hc3** are used in the local strategy of the **t-heuristic-classification** task. The control process focuses only on the characteristics indicated by the foci. In our example, the focus **f-patient** privileges processes which operate on the concept **c-patient** Foci **f-hc**, **f-hcl**, **f-hc2**, **f-hc3** privilege some processes directly by their names.

- **Strategies**

  The axiom defining the body of a strategy is an equation stating the equality between the operator **body** applied to the name of the strategy and an expression in the strategy body language. The process module shown in Figure 8 contains the specification of the local strategies of the t-heuristic-classification task.

[9] It might also have been possible to describe the task problem-solving knowledge by the use of a non-deterministic process: heuristic-classification ⊕ match;refine ⊕ match which expresses an alternative between the heuristic-classification process and its final part, and to define a local strategy to make the choice between these three processes.

3.2.2. The semantics of process modules

The specification **Process** gives the global semantics to the set of all the process modules (Figure 11). This specification is organised in two layers: **Process0** and **Process**. First, **Process0** (Figure 10) defines the constructors of processes and all their static characteristics of all the application processes. Then, **Process** (Figure 9) defines for each process its application to data; this part requires **Process0** through a "use". The sort **exp-bool** is also defined in **Process**.

The specification **Process** contains the signatures and axioms of all the process modules. Its signature has two parts: the first is constant and common to all the processes specifications (δ, ?, *, $\oplus$, ;, application operators); the other is precisely the one defined by the KE, and is limited to the declarations of atomic, composed and task processes. The equations of the specification process are also divided into two parts: a constant part assigns a meaning to the operators of the signature constant part; a variable part defined by the KE precises the semantics of the declared processes.

Process also includes the specification of the strategic knowledge. It uses the specifications **Focus** (Appendix, Figure 14), **Strategy** (Appendix, Figure 13) and **Task** (Appendix, Figure 15) which respectively give the semantics of foci, strategy and tasks. The strategies administrator behaviour is also specified within **Process** (appendix, Figure 16).

4 Comparison with other formal languages

This comparison is based on classical language features proposed in software engineering (Gaudel, 1994), or KE literature (Aben, 1995), such as spectrum, orientation, institution, semantics, methods.

- *Domain spectrum*

  Most existing formal languages are based on a task-oriented approach, more particularly on the KADS expertise model. An exception is DESIRE (van Langevelde et al., 1992, 1993) which is based on a multi-agent paradigm. Thus, although supposed to be general, in practice these languages are more suited to a specific type of KBS, respectively either task- or agent-based systems. The TASK conceptual modelling approach combines a task-oriented approach with a distributed approach. TFL permits us to specify systems which may require the activation of several Task-Modules (agents) to meet the global expected functionality. Moreover, it is possible to specify not only one problem-solving method (as in the task layer of KADS), but also several alternate methods available for achieving the competence of each Task-Module. Moreover, TFL enables strategic reasoning, both within a Task-Module (dynamic generation of plans) and more generally between several processes and Task-Modules, which can be dynamically selected and activated. The recursive definition of strategies in TFL permits any number of control levels to be defined. The meta-system control processes are also specified in TFL. TFL is a language suited to the specification of reflective KBSs with flexible problem-solving. For all these reasons, TFL offers a wider spectrum than KADS and DESIRE languages.
- *Method spectrum*

  Some existing languages focus on the requirement analysis stage, such as $(ML)^2$ (van Harmelen & Balder, 1992), $K_{BS}SF$ (Jonker & Spee, 1992) and Qil (Aitken et al., 1992). TFL is intended to cover the design stage as well, as do KARL (Fensel et al., 1991) and DESIRE (van Langevelde et al., 1992, 1993). Concerning the design step, it can be expected that many results can be inherited from software engineering work and experiences with ADTs in this area. We are studying this direction, and work is in progress to prove the correctness of an implemented KBS.
- *Institution*

  The institution of most formal languages in KE is some type of logic, except for $K_{BS}SF$ which is based on algebra. TFL is based on algebra, since it is based on the algebraic specification language PLUSS.
- *Orientation*

  While most languages are model-oriented, except for $K_{BS}SF$ which is behaviour-oriented, TFL can be considered rather as property-oriented, since the axioms in ADT are a natural means of

spec Process-Name
sorts process-name
operations
for each process 'p' (see section 3.2.1)
p : $\mapsto$ process-name

spec Task-Name
sorts task-name
operations
for each task 't' (see section 3.2.1)
t : $\mapsto$ task-name

spec Process
use Process0
sorts process
operations
$— :: —$: process $\times$ data $\mapsto$ data
$\llcorner — <::> —$: exp-bool $\times$ state $\mapsto$ bool
apply-inference : relation $\mapsto$ process
axioms
$\psi <::> D[STATE] = true \Rightarrow \psi? :: D = D$ (a_1)
$\psi <::> D[STATE] = false \Rightarrow \psi? :: D = \emptyset$ (a_2)
$\delta :: D = \emptyset$ (a_3)
$p; q :: D = q :: (p :: D)$ (a_4)
$p :: \emptyset = \emptyset$ (a_5)
$p \oplus q :: D =$ manage-strategy;process-choice$(p,q) :: D$ (a_6)

for each task 't' (see section 3.2.1)
t : process $:: D =$
activate-strategy(strategies(t : task));manage-strategy;
competence-satisfaction(t : task) $:: D$ (a_7)

$I1 \in D[STATE] \wedge R(I1,I2) \Rightarrow$
apply-inference$(R) :: D = D[STATE \backslash I2 \; \& \; D[STATE]\;]$
$\neg(I1 \in D[STATE] \wedge R(I1,I2)) \Rightarrow$ apply-inference$(R) :: D = D$ (a_8)

$false <::> D[STATE] = false$
$true <::> D[STATE] = true$

for each boolean expression 'ψ'
$\psi <::> D[STATE] =$ *"'state' boolean expression"*

axioms defining application
$AX^{inference}_{application}$ $AX^{composed_process}_{application}$ (see section 3.2.1)

axioms defining strategy administrator
$AX_{strategy_administrator}$ *(developped Fig. 16)*

where
D : data
p, q, r : process
$I1, I2$: information
R : relation

Figure 9 Semantics of processes: the "Process" layer

spec Process0
  use Data, Process-Name, Task, Set-Information, Set-Concept, Set-Resource,
  Set-Process-Name, Set-Task-Name, Set-Strategy, Set-Foci
  sorts process, exp-bool
  operations

—	: process-name $\mapsto$ process
δ	: $\mapsto$ process
—	: task : $\mapsto$ process
—?	: exp-bool $\mapsto$ process
—$\star$	: process $\mapsto$ process
— $\oplus$ —	: process $\times$ process $\mapsto$ process
— ; —	: process $\times$ process $\mapsto$ process

pre,post : process $\mapsto$ exp-bool
input,output : process $\mapsto$ set-concept
resource : process $\mapsto$ set-resource
process : process $\mapsto$ set-process-name
tasks : process $\mapsto$ set-task-name

process-choice : process $\times$ process $\mapsto$ process
activate-strategy : task-name $\mapsto$ process
competence-satisfaction : situation $\times$ set-process-name $\mapsto$ process
manage-strategy : $\mapsto$ process
eval-stategy : strategy-name $\mapsto$ process
eval-stategies, stop-strategies : set-strategy-name $\mapsto$ process

false, true : $\mapsto$ exp-bool
for each another boolean expression 'ψ'
ψ : $\mapsto$ exp-bool

predicates
— $\in$ — or — : process $\times$ process $\mapsto$ process

preconditions
$p; q$ **is defined iff** $\mathrm{post}(p) <::> D = true \Rightarrow \mathrm{pre}(q) <::> D = true$

axioms

$(p \oplus q) \oplus r = p \oplus (q \oplus r)$ (A)
$p \oplus q = q \oplus p$ (B)
$p \oplus p = p$ (C)
$p \oplus \delta = p$ (D)
$(p; q); r = p; (q; r)$ (E)
$\delta; p = \delta$ (F)
$(p \oplus q); r = (p; r) \oplus (q; r)$ (G)
$r; (p \oplus q) = (r; p) \oplus (r; q)$ (H)
$p\star = p \oplus p; p\star$ (I)

$p \in p$ or q = true (c_1)
$q \in p$ or q = true (c_2)
$\neg r = p \wedge \neg r = q \Rightarrow p \in q$ or q = false (c_3)
process-choice(p,q) = process-choice(q,p) (c_4)
process − choice(p,q) $\in p$ or q = true (c_5)

axioms defining characteristical properties of process
$\mathbf{AX}_{\mathrm{characteristics}}$ *"developed Fig. 12"*

where
p, q, r : process
ψ : bool

Figure 10 Semantics of processes: the "Process0" layer

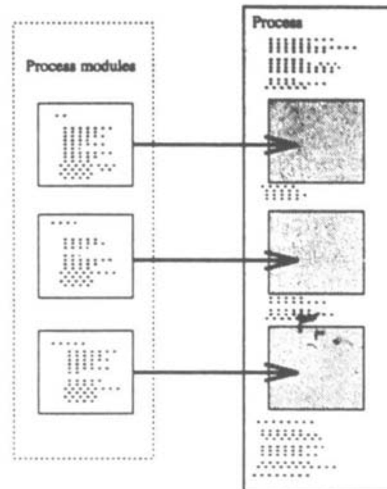

Figure 11 The global specification of the process-solving knowledge

specifying properties of operations. However, since TFL particularly focuses on the behaviour specification, it can also be viewed as behaviour-oriented.

- *Semantics*
  Formal languages in KE either have a denotational or an operational semantics (Fensel & van Harmelen, 1994). TFL has a denotational semantics since it is based on the PLUSS algebraic specification language. The *stratified loose semantics* of PLUSS is of the "set of models" type. It can be considered as extending both the initial and the set of models approach so as to enable one to provide a specification module with some semantics (Bidoit, 1989). Its advantage is to allow us to structure large specifications into smaller units, and to provide parametrisation mechanisms.
- *Method*
  Most languages provide horizontal structuring mechanisms. Indeed, for instance $(ML)^2$, $K_{BS}SF$, KARL, DESIRE enable us to structure the specification into several modules with precise links based on their expertise model. For instance, views and terminators are mechanisms to connect domain and inference layers in KARL. DESIRE is an example which offers mechanisms to decompose modules into smaller modules. Most of them, however, do not provide vertical structuring mechanisms. No existing language provides an elaborate refinement calculus. Only Aben (1995) has recently proposed interesting directions for refinement calculus in KE. Modularity is an essential aspect of the PLUSS language (Capy, 1987; Bidoit, 1989) used to develop TFL. Its powerful modularity mechanisms provide TFL with interesting properties. First, PLUSS provides several enrichment constructs (e.g., the import primitive *use*) which enable us to structure a specification into modules. Second, it also offers parametrisation and instantiation mechanisms. Moreover, it favours progressive development of specifications, since it allows us to define two types of specification. Completed specifications, with a fixed class of models, are the "implementable" modules, while *sketch* and *draft* are specifications under development. This distinction is one main originality of PLUSS. It implies important consequences on the development process of a specification (Bidiot, 1989). During the elaboration stage of the specification, the semantics of the specification is not yet completely fixed. Then, draft and sketch modules enable us to build successive specifications (not yet implementable) by progressive refinements. When the specification is considered to be satisfying, its semantics must be fixed, and then it is translated into an "achieved" specification (implementable). This constraint is a necessary condition to facilitate reuse. All these mechanisms, which have been helpful when applying TFL on examples like Sisyphus I or Sisyphus II (Pierret-Golbreich, 1996b), offer interesting perspectives for the design stage. For example, the specification of goals by progressive refinements presented in (Pierret-Golbreich, 1996b) is based on this facility: VT-

parametric design and the Office allocation examples have been built by stepwise refinement of the generic draft "assignment". It should be noticed that vertical refinement is of primary importance in ADTs. "One of the main interests of algebraic specifications is the possibility of developing a program by starting from a high-level abstract specification and giving successive, more and more detailed representations of this specification" (Gaudel, 1990). Interesting results may surely be expected from ADTs for formal methods in KE, but more work is still needed to precisely investigate its potential application to TFL.

Conclusion

This specification has largely been motivated by the wish to investigate the possibilities of using software engineering formal languages to specify flexible KBS and, in particular, their dynamic behaviour. In this work, several results interesting from a knowledge engineering point of view have been obtained:

- Strategic knowledge and meta-system specification: this is one main contribution of the work. Indeed, a control formal specification has never been undertaken until now in the framework of task-oriented approaches. The single formalism which offered such a possibility of specifying strategic reasoning was DESIRE. TFL is more understandable and easier to use than DESIRE. Indeed, first it belongs to task-oriented approaches, which are generally the most preferred for specifying KBSs at the knowledge level. Second, TFL specification of the meta-system is expressed at the knowledge level, while DESIRE global control specification is based on a rule formalism. Finally, like DESIRE, TFL offers the possibility of describing any level of strategies. The basic idea is to consider the resolution problem as the application of processes to data. A process corresponds to a deterministic or non-deterministic process composition. The role of the meta-system controlling the reasoning is to eliminate the non-deterministic processes, replacing them by dynamically built fixed processes, thanks to the specified strategies.
- State representation: the state (i.e. the current case data) are specified as information in the Data module, in contrast to the domain knowledge (concepts, relations, domain structures) which is specified as specific data types. The resolution process is viewed as the completion activity of the initial set of information by use of the different categories of knowledge (domain, problem-solving, strategic knowledge), in particular applying relevant processes to data. Thus, the key operator in TFL is the *application* operator through which processes are applied to the state (data) to reach a solution state.
- Global system specification: TFL is dedicated to the specification of the global system behaviour. In contrast to other languages, which only aim at specifying the object system, that is the knowledge of particular applications, TFL aims at specifying both the object *and* meta-system controlling the resolution.
- Benefits of a single formalism: an interesting feature of the specification is its uniformity, which allows us to avoid the difficulties concerning the link between different formalisms.
- Precise semantics: all the TASK primitives have got a detailed and precise semantics in TFL.
- Benefits of using ADT:
  —abstraction: the specifications obtained are *abstract*, describing the "idea" of the KE about its system independently of any implementation.
  —modularity: the specification is *modular*. A complex specification is built by combining diverse modules which are less complex.
  —semantics: the use of the process modules makes much more easy the design of a KBS specification. It allows us to write simple specifications and, at the same time, to benefit from a clear and precise *semantics* for the whole system, thanks to their inclusion in classical ADTs.
  —mathematical grounds of the ADT: the *axiomatic* aspect of the algebraic specifications makes them suitable for prototyping and for proofs, because of the natural link existing between the algebraqic specifications and rewriting systems.

$\mathbf{AX}_{\text{characteristics}}$ =

pre(p; q) = pre(p)	(c;₁)
post(p; q) = post(q)	(c;₂)
input(p; q) = input(p) & input(q) & ∅	(c;₃)
output(p; q) = output(p) & output(q) & ∅	(c;₄)
resources(p; q) = resources(p) & resources(q) & ∅	(c;₅)
processes(p; q) = processes(p) & processes(q) & ∅	(c;₆)
tasks(p; q) = tasks(p) & tasks(q) & ∅	(c;₇)
pre(p ⊕ q) = pre(p) ∨ pre(q)	(c⊕₁)
post(p ⊕ q) = post(p) ∨ post(q)	(c⊕₂)
input(p ⊕ q) = input(p) & input(q) & ∅	(c⊕₃)
output(p ⊕ q) = output(p) & output(q) & ∅	(c⊕₄)
resources(p ⊕ q) = resources(p) & resources(q) & ∅	(c⊕₅)
processes(p ⊕ q) = processes(p) & processes(q) & ∅	(c⊕₆)
tasks(p ⊕ q) = tasks(p) & tasks(q) & ∅	(c⊕₇)
pre(p⋆) = pre(p)	(c⋆₁)
post(p⋆) = post(p)	(c⋆₂)
input(p⋆) = input(p)	(c⋆₃)
output(p⋆) = output(p)	(c⋆₄)
resources(p⋆) = resources(p)	(c⋆₅)
processes(p⋆) = processes(p)	(c⋆₆)
tasks(p⋆) = tasks(p)	(c⋆₇)
post(p⋆)⇒ pre(p⋆)	(c⋆₈)
pre(ψ?) = ψ	(c?₁)
post(ψ?) = ψ	(c?₂)
input(ψ?) = ∅	(c?₃)
output(ψ?) = ∅	(c?₄)
resources(ψ?) = ∅	(c?₅)
processes(ψ?) = ∅	(c?₆)
tasks(ψ?) = ∅	(c?₇)
pre(δ) = false	(cδ₁)
post(δ) = false	(cδ₂)
input(δ) = ∅	(cδ₃)
output(δ) = ∅	(cδ₄)
resources(δ) = ∅	(cδ₅)
processes(δ) = ∅	(cδ₆)
tasks(δ) = ∅	(cδ₇)
pre(t : process) = pre(t : task)	(ct₁)
post(t : process) = post(t : task)	(ct₂)
input(t : process) = input(t : task)	(ct₃)
output(t : process) = output(t : task)	(ct₄)
resources(t : process) = resources(t : task)	(ct₅)
processes(t : process) = processes(t : task)	(ct₆)
tasks(t : process) = t : task & tasks(t : task) & ∅	(ct₇)

characteristics for task application

pre(competence-satisfaction(t : task) ⇒ pre(t : process)	(t₁)
post(competence-satisfaction(t : task) ⇒ post(t :process)	(t₂)
resources(t :process) ∈ resources(competence-satisfaction(t : task))	(t₃)
input(t : process) ∈ input(competence-satisfaction(t : task))	(t₄)
output(t : process) ∈ output(competence-satisfaction(t : task))	(t₅)
processes(t : process) ∈ processes(competence-satisfaction(t :task))	(t₆)
tasks(t : process) ∈ tasks(competence-satisfaction(t : task))	(t₇)

axioms defining characteristical properties

$\mathbf{AX}^{\text{tasks}}_{\text{characteristics}}$ $\mathbf{AX}^{\text{inferences}}_{\text{characteristics}}$ *(see section 3.2.1)*

Figure 12 Semantics of characteristics

```
spec Strategy-Name
   sorts strategy-name
   operations
   for each strategy 's_name'
        s_name ↦ strategy-name

spec Set-Strategy-Name as
        set( item ⇒ strategy-name)
```

```
spec Strategy-Body
   use Strategy-name, Set-Strategy-Name, Focus
   sorts strategy-body
   operations
        ε : strategy-body
        body : strategy-name ↦ strategy-body
        ( if — — ) : exp-bool × strategy-body ↦ strategy-body
        ( loop — — ) : exp-bool × strategy-body ↦ strategy-body
        — ; — : strategy-body × strategy-body ↦ strategy-body
        post-strategy, stop-strategy : set-strategy-name ↦ strategy-body
        post-focus, stop-focus : set-focus ↦ strategy-body
   axioms
        ¬ body( S) = X;poststrategy( Sn)                              (a)
        ¬ body( S) = X;postfocus( Fn)                                 (b)
        ¬ body( S) = X;stopstrategy( Sn)                              (c)
        ¬ body( S) = X;postfocus( Fn)                                 (d)
        ¬ body( S) = X;( if cond Z)                                   (e)
        ¬ body( S) = X;( loop cond Z)                                 (f)
        for each strategy name 'Sname'
        body(Sname) = ...                                             (g)
   where
        S : strategy-name
        Sn : set-strategy-name
        Fn : set-focus
        X, Y, Z : strategy-body
```

```
spec Strategy as  Record_of(   < NAME : strategy-name >,< BODY : strategy-body >,
                               < MEM_S : set-strategy-name >, < MEM_F : set-focus-name >)
   sorts strategy
   operations  name-to-strategy : strategy-name ↦ strategy
   axioms
        name-to-strategy(Sname) = < < NAME : Sname >,
             < BODY : body( Sname) >, < MEM_S : ∅ >, < MEM_F : ∅ > >  (a)
   where Sname : strategy-name
```

```
spec Set-Strategy as  set( item ⇒ strategy)
   operations
        names-to-strategies : set-strategy-name ↦ set-strategy
        strategies-to-names : set-strategy ↦ set-strategy-name
   axioms
        strategies-to-names(∅) = ∅                                    (a)
        S[NAME] = Sname ⇒
             strategies-to-names(S & RS) = Sname & strategies-to-names(RS)   (b)
        names-to-strategies(∅) = ∅                                    (c)
        names-to-strategies(Sname & RSn) =
             name-to-strategy(Sname) & names-to-strategies(RSn)       (d)
   where
        S : strategy
        Sname : strategy-name
        RS : set-strategy
        RSn : set-strategy-name
```


Figure 13 Semantics of strategies

```
spec Focus
  use
      Set-Resource, Set-Concept, Exp-Bool, Set-Task-Name, Set-Process-Name
  sorts focus
  operations
      input, output : focus ↦ set-concept
      resource : focus ↦ set-resource
      process : focus ↦ set-process-names
      tasks : focus ↦ set-task-name
      pre, post : focus ↦ exp-bool
      for each focus name 'f' : (see section 3.2.1)
      f : ↦ focus

  axioms
      for each focus name 'f' : (see section 3.2.1)
      input(f) = ...
      output(f) = ...
      pre(f) = ...
      post(f) = ...
      resources(f) = ...
      processes(f) = ...
      tasks(F) = ...
```

Figure 14 Semantics of foci

```
spec Task
  use
      task-name, Set-Strategy-Name, Set-Task-Name, Set-Process-Name, Set-Resource, Exp-Bool
  sorts task
  operations
      — : task-name ↦ task
      pre, post : task ↦ exp-bool
      input, output : task ↦ set-concept-name
      resource : task ↦ set-resource
      tasks : task ↦ set-task-name
      process : task ↦ set-process-name
      strategies : task ↦ set-strategy-name
  axioms
      for each task 't' (see section 3.2.1)
      input(t) = ...
      output(t) = ...
      pre(t) = ...
      post(t) = ...
      resources(t) = ...
      processes(t) = ...
      tasks(t) =...
      strategies( t) = ...
```

Figure 15 Specification of tasks

$\mathbf{AX}_{strategy_administrator}$ =

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = $s_n \wedge$
s[BODY] = (if *cond* s_1); s_2 $\wedge$ *cond* < :: > D[STATE] = true $\Rightarrow$
eval-strategy(s_n) :: D =
eval-strategy(s_n) :: D[STRATEGIE\ s[BODY\ s_1; s_2] & *strats*] (ad_1)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = $s_n \wedge$
s[BODY] = (if *cond* s_1); s_2 $\wedge$ *cond* < :: > D[STATE] = false $\Rightarrow$
eval-strategy(s_n) :: D =
eval-strategy(s_n) :: D[STRATEGIE\ s[BODY\ s_2] & *strats*] (ad_2)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = $s_n \wedge$
s[BODY] = (loop *cond* s_1); s_2 $\wedge$ *cond* < :: > D[STATE] = true $\Rightarrow$
eval-strategy(s_n) :: D =
eval-strategy(s_n) :: D[STRATEGIE\ s[BODY\ s_2] & *strats*] (ad_3)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = $s_n \wedge$
s[BODY] = (loop *cond* s_1); s_2 $\wedge$ *cond* < :: > D[STATE] = false $\Rightarrow$
eval-strategy(s_n) :: D =
eval-strategy(s_n) :: D[STRATEGIE\ s[BODY\ s_1; (loop *cond* s_1); s_2] & *strats*] (ad_4)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = s_n $\wedge$ s[BODY] = post-strategie(sts_n); s_1 $\Rightarrow$
eval-strategy(s_n) :: D =
D[STRATEGIE\ s[BODY\ s_1][MEM_S\ sts_n & s[*MEM_S*]]
& names-to-strategies(sts_n) & *strats*] (ad_5)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = s_n $\wedge$ s[BODY] = post-focus(f_n); s_1 $\Rightarrow$
eval-strategy(s_n) :: D =
D[STRATEGIE\ s[BODY\ s_1][MEM_F\ f_n & s[*MEM_F*]] & *strats*]
[FOCUS\ f_n & D[FOCUS]] (ad_6)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = s_n $\wedge$ s[BODY] = stop-focus(f_n); s_1 $\Rightarrow$
eval-strategy(s_n) :: D =
D[STRATEGIE\ s[BODY\ s_1][MEM_F\ f_n & s[*MEM_F*]] & *strats*]
[FOCUS\ remove(f_n, D[FOCUS]] (ad_7)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = s_n $\wedge$ s[BODY] = stop-strategie(sts_n); s_1 $\Rightarrow$
eval-strategy(s_n) :: D = stop-strategies(sts_n) :: (D[STRATEGIE\ s[BODY\ s_1] & *strats*]) (ad_8)

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = s_n $\wedge$ s[BODY] = ϵ $\Rightarrow$
eval-strategy(s_n) :: D = stop-strategies(s_n & $\emptyset$) :: D (ad_9)

eval-strategies($\emptyset$) :: D = D (ad_{10})
eval-strategies(s_n & rs_n) = eval-strategies(s_n); eval-strategies(rs_n) (ad_{11})

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = s_n $\wedge$ s[MEM_S] = st_n & rst_n $\Rightarrow$
stop-strategies(s_n & $\emptyset$) :: D =
stop-strategies(rst_n); stop-strategies(st_n & $\emptyset$);
stop-strategies(s_n& $\emptyset$) :: (D[STRATEGIES\ s[MEM_S\ $\emptyset$] & *strats*]) (ad_{12})

D[STRATEGIES] = s & *strats* $\wedge$ s[NAME] = s_n $\wedge$ s[MEM_S] = $\emptyset$ $\Rightarrow$
stop-strategies(s_n & $\emptyset$) :: D =
D[FOCUS\ remove(s[MEM_S], D[FOCI])][STRATEGIES\ *strats*] (ad_{13})

stop-strategies($\emptyset$) :: D = D (ad_{14})
stop-strategies(s_n & rs_n) = stop-strategies(s_n & $\emptyset$); stop-strategies(rs_n) (ad_{15})

manage-strategy :: D = eval-strategies(strategies-to-names(D[STRATEGIES])):: D (ad_{16})
activate-strategy(sts_n) :: D = D[STRATEGIES\ names-to-strategies(sts_n) & D[STRATEGIES]] (ad_{17})

Figure 16 Strategic knowledge administration

- An original model of control: from the modelling viewpoint, this approach offers an original model of control since resolution processes call the control when and *only* when a choice has been made. This result is important, considering that it is acknowledged that in a KBS, most of the time is used in the control activity (Hayes-Roth, 1985). The model combines hierarchical and opportunistic approaches. The reflexivity of the model allows us to gain uniformity and simplicity in the knowledge modelling.

This study opens several prospects concerning verification and validation and reuse. The classical algebraic approach has demonstrated its benefits in the fields of the specification of computational systems, the proof about the properties of a specification, concerning the implementation and the validity of implementation. These directions are currently being investigated for KBSs. For instance, the correction of methods w.r.t problems specification proof is studied in Pierret-Golbriech (1996a). The issue of genericity and reuse based on formal specifications is presented in (Pierret-Golbreich (1996b), which shows how formal methods can support the definition of generic formal specifications and the development of a formal KBS specification by reuse.

Appendix

The following appendix shows axioms which complete the specification of the TFL language. The static semantics of the processes is defined by the axioms of Figure 12. For each type of composed-process, an axiom defines the value of its characteristics (input, output, resources, etc.) from its components. Similarly, for each task and inference, the axioms defining the static semantics are given here. Figure 13 gives the semantics of strategies. A strategy is composed of a strategy name and a strategy body. The current state of a strategy and the current foci activated by it are placed in its fields **MEM_S** and **MEM_F**. Figure 16 shows the specification of the strategy administrator, which operates on strategies as a kind of "rewriting rules" tool. Each axiom expresses the changes of current active foci and strategies for a particular type of strategy body. The two other figures present the specification of foci (Figure 14) and task (Figure 15). In these figures, the axioms give the definition of the different characteristics.

References

Aben, M, 1993. "Formally specifying reusable knowledge model components" *Knowledge Acquisition* **5** 119–141.

Aben, M, 1995. *Formal methods in knowledge engineering*. PhD thesis, Universiteit van Amsterdam.

Albert, P, Corby, O, Gobinet, P and Neveu, B, 1992. Langage de spécification de la communication et du contrôle dans un blackboard, Rapport intermédiaire No III.1 du contrat 1.91.E117, DRET-ILOG-INRIA.

Aitken, S, Reichgelt, H and Shadbolt, N, 1992. Representing KADS models in QI. Technical report, AI Group, May 27–31, University of Nottingham.

Breuker, J and Van de Velde, W, 1994. *Common KADS Library for Expertise Modelling*. IOS Press.

Bidiot, M, 1989. Pluss, un langage pour le développement de spécifications algébriques modulaires. PhD thesis, Université de Paris-Sud.

Brachman, RJ and Schmolze, JG, 1985. An overview of the KL-ONE knowledge representation system. *Cognitive Science* **9** 171–216.

Capy, F, 1987. ASSPEGIQUE: un environnement d'exeptions... Une semantique operationnelle des er-algebres, formalisme prenant en compte les exeptions. un environnement integre de specification algebrique: asspegique. PhD thesis, Université Paris-Sud.

Chandrasekaran, B, 1988. Generic tasks as building blocks for knowledge-based-reasoning: the diagnosis and routine design exemples. *Knowledge Engineering Review* **3** (3).

McDermott, J, 1988. Preliminary steps towards a taxinomy of problem solving methods. In: S. Marcus, (ed.), *Automating Knowledge Acquisition for Expert Systems*, 225–255, Kluwer Academic.

Fensel, D, Angele, J and Landes, D, 1991. "A knowledge acquisition and representation languages". In: Proc. *Expert Systems and their Applications, 11th International Workshop, Conference Tools, Techniques and Methods*, Avignon, France, May 27–31.

Fensel, D, 1995. "Formal specification languages in knowledge and software engineering". *The Knowledge Engineering Review* **10** (4) 361–404, December.

Fensel, D, 1995. *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic.
Fensel, D and van Harmelen, F, 1994. "A comparison of langages which operationalise and formalise KADS models of expertise". *The Knowledge Engineering Review*.
Gaudel, MC, 1884. "A first introduction to PLUSS". In: *METEOR Report*, 493–510, December.
Gaudel, MC, 1985. "Towards structured algebraic specifications". In *ESPRIT'85 Status Report*, North Holland, September 23–25.
Gaudel, MC, 1990. *Algebraic specifications*. Rapport 557, LRI, University Paris Sud.
Gaudel, MC, 1994. "Formal specification techniques, invited state-of-the-art report, extended abstract". In: *IEEE-ACM International Conference on Software Engineering*, 223–227.
Gaudel, MC, 1992. "Test selection based on ADT specification". In: *IWPT92*, Wiley.
Harel, D, 1984. *Handbook of Philosophical Logic, volume II: Extensions of Classical Logic*, Chapter "Dynamic Logic", Reidel.
Van Harmelen, F and Balder, F, 1992. "$(ML)^2$: A formal language for KADS models of expertise". *Knowledge Acquisition* **4** 127–161.
Hayes-Roth, B, 1985. "A blackboard architecture for control". *Artificial Intelligence* **26** 251–321.
Jonker, W and Spee, J, 1992. "Yet another formalisation of KADS conceptual models". In: T Wetter *et al.* (eds), *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, 211–229.
Kaplan, S, 1987. Spécification algébrique de types de données à accès concurrent. PhD thesis, Université de Paris-Sud.
Linster, M, Karbach, W, Voß, A and Walther, J, 1992. "An analysis of the role of operational modelling languages in the development of knowledge-based systems". In: *Proceedings of the 2nd Japaneses Knowledge Acquisition for Knowledge-Based Systems Workshop*, Hatayoama, Japan.
Marre, B, 1991. "Toward automatic test data set selection using algebraic specifications and logic programming". In: *Proceedings of the Eighth International Conference*, 202–221, MIT Press.
Marre, B, 1991. Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques. PhD thesis, Université de Paris-Sud.
Musen, MA, 1989. *Automated Generation of Model-Based Knowledge-Acquisition Tools*. Research Notes in Artificial Intelligence, Pitman.
Newell, A, 1982. "The knowledge level". *Artificial Intelligence* **18** 35–418, 87–1273.
Orejas, F, Navarro, M and Sanchez, A, 1992. Algebraic implementation of abstract data types: a survey. In: *Proceedings 8 WADT 3rd COMPASS Workshop*, Dourdan, France
Pierret-Golbreich, C, 1994. "Task model: a framework for the design of models of expertise and their operationalization". In: *Proceedings Knowledge Acquisition Workshop*, Banf, Canada.
Pierret-Golbreich, C. 1996. "Correction of methods w.r.t problems specifications". In: *ECAI-96 Workshop "Validation, Verification and Refinements of KBS"*.
Pierret-Golbreich, C, 1996. "Modular and reusable specifications in knowledge engineering: formal specification of goals and their development". In: *6th Workshop on Knowledge Engineering Methods and Languages*, Paris, France.
Pierret-Golbreich, C and Talon, X, 1994. "Specifications formelles des connaissances pour l'acquisition: une approche basée sur les types abstraits algébriques". Unpublished.
Steels, L, 1990. "Components of expertise". *AI Magazine*.
Terry, A, 1983. "The crysalis project: Hierarchical control of production systems. Technical Report HPP-83-19, Stanford University.
Treur, J and Wetter, T, 1993. *Formal Specification of Complex Reasoning Systems*. Ellis Horwood.
van Langevelde, IA, Philipsen, AW and Treuer, J, 1992. "Formal specification of compositional architectures". In: *Proceedings of the 10th European Conference, ECAI'92*, 272–276, Wiley.
van Harmelen, F and Fensel, D, 1995. "Formal methods in knowledge engineering". *The Knowledge Engineering Review* **10** (4) 345–360, December.
van Langevelde, I, Philipsen, A and Treur, J, 1992. "Formal specification of compositional architectures". In: *Proceedings of the 10th European Conference on Artifical Intelligence (ECAI 92)*, Vienna, Austria.
van Langevelde, I, Philipsen, A and Treur, J, 1993. "Formal specification of complex reasoning systems" In: *A Compositional Architecture For Simple Design Formally Specified in DESIRE*, Ellis Horwood.
Wielinga, B, Van de Velde, W, Schreiber, G and Akkermans, H, 1993. *Expertise Model Definition Document*, University of Amsterdam.
Wetter, T and Schmidt, W, 1991. "Formalisation of the KADS interpretation models". In: *Proceedings of the 8th Conference of the Society for the Study of Artificial Intelligence and Simulation of Behavior (AISB'91)*, Springer-Verlag.
Wielinga, B, Schreiber, G and Breuker, J, 1992. "KADS: A modelling approach to knowledge engineering". *Knowledge Acquisition Journal* **4** (1) 1–162.