Cokace: A Centaur-based environment for CommonKADS Conceptual Modelling Language

Olivier Corby and Rose Dieng[1]

Abstract.
In order to help the knowledge engineer during knowledge modelling phase, we developed an environment for the conceptual modelling language CML offered by the CommonKADS methodology. This environment, called Cokace, was designed using Centaur, a programming environment generator, that was usually exploited for building environments dedicated to software engineering languages. Thanks to Centaur, Cokace provides the knowledge engineer with structured edition, static validation and dynamic interpretation of CML expertise models. Cokace allows the knowledge engineer to simulate a reasoning on CML expertise models, and enables verification and evaluation of such expertise models before implementation of the final knowledge-based system. This work illustrates an example of the benefits knowledge engineering can get from well established techniques and tools available in software engineering.

1 INTRODUCTION

Knowledge modelling is considered as a crucial activity for the development of a knowledge based system (KBS), or for expertise capitalization in a workplace. CommonKADS [22, 5, 31], COMMET [25, 26], respective fruits of the Esprit projects KADS-II and CONSTRUCT, are well known examples of methodologies enabling knowledge modelling at the knowledge level [18].

Validation of the modelled knowledge is a significant and difficult problem: if validation of static knowledge is possible in some knowledge acquisition methods or tools, validation of dynamic knowledge (for example by simulation of expertise models) is seldom offered [28]. Under the term "validation", Laurent distinguishes verification (i.e. "objective validation w.r.t. a formal specification") and evaluation (i.e "interpretative validation w.r.t. a formal but heuristic measurement process and a subjective threshold") [17]. Clearly, the knowledge engineer (KE) should not wait until implementation in order to validate and run knowledge models. So, validating tools would be useful during knowledge modelling phase, in particular tools able to work at the knowledge level. KREST, the COMMET-supporting tool, allows different kinds of validation after each significant phase of the KBS development process [7, 9]. The motivations of our research reported here were to provide validation support for expertise modelling: more precisely, we aim at offering an environment allowing the KE to build expertise models, with adequate validation capabilities, in the framework of CommonKADS.

The CommonKADS methodology [22] proposes the development of several models: organisation, task, agent, communication, expertise, design. It promotes the reuse of generic models through a generic modelling library [5]. It provides a language, CML (Conceptual Modelling Language), for describing expertise models at a conceptual level [21]. Whereas KADS Tool [10] and the CommonKADS Workbench [11] are dedicated to CommonKADS, Open-KADS [6] offers a help within the KADS-I framework. The KACTUS project [23] proposes, among others, an editing environment for CML. Our intended environment will also rely on CML, but will focus on validation capabilities.

Formal specification and verification problems have been studied for a long time by software engineering (SE) researchers: several techniques, methods and tools are now available. Therefore, our approach was to exploit a well established software engineering tool, offering interesting verification features: we chose Centaur, a powerful generator of programming environments.

After a presentation of the Centaur system, we will indicate how we exploited it to build the Cokace (COmmonKAds-CEntaur) environment dedicated to the CML language. We will show how Cokace can help the KE to design, check and run CommonKADS expertise models described in CML. We will indicate some kinds of validation allowed by Cokace and conclude by a comparison with related research and a description of further work.

2 EXPLOITATION OF CENTAUR

2.1 Principles of Centaur

Centaur is a generic interactive environment [4, 12], developed in the framework of two Esprit projects: "Generation of Interactive Programming Environments", GIPE and GIPE II. It produces a specific programming environment for a language, when given the specification of its syntax and semantics. The environment generated by Centaur includes a structured editor, a type-checker, an interpreter for the language, and possibly other tools. The user interacts with the generated environment by means of a graphic interface also generated by Centaur. Powerful and enabling elegant programming, Centaur was used to build environments for programming languages such as ADA, C, Fortran, Pascal, etc. The generated environment can then be exploited by the user for building programs in the language, such programs being checked syntactically and semantically. Therefore, we considered CML as a classic language, so as to take benefit of Centaur verification capabilities. Thus, we focused on the "language" aspect, that is important for the development of a software (a KBS being a specific software). The "knowledge" aspect - specificity of knowledge engineering w.r.t. SE - was already tackled in the design of CommonKADS and in the choice of CML language primitives (e.g. expertise model, task, inference, domain model, ontology, etc.).

The Centaur system is composed of: (1) a database component, supplying standardized representation and access to formal objects

[1] INRIA, Sophia-Antipolis, Acacia Project, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex, FRANCE

ECAI 96. *12th European Conference on Artificial Intelligence*
Edited by W. Wahlster
Published in 1996 by John Wiley & Sons, Ltd.

and their persistent storage, (2) a logical engine used to execute formal specifications, (3) an object-oriented man-machine interface, giving easy access to the system's functions. The architecture of Centaur is based on a decomposition into three classes of functions: (1) a kernel for representation and manipulation of structured objects such as abstract syntax trees, (2) a specification level that supports syntactic and semantic aspects of languages, (3) a user interface, for interactive communication between Centaur and its users. Centaur offers several formalisms: (1) Metal to define the syntax of the language for which an environment is intended to be generated, (2) PPML to specify how to pretty-print the programs of the language, (3) Typol to specify the semantics of the language.

A more detailed description can be found in [4, 12]. The next subsections will illustrate how we exploited Centaur to build Cokace, our environment dedicated to the CML language.

2.2 Definition of the language syntax

The language syntax must be defined by the programmer by means of the Metal formalism. A Metal specification consists of three parts: the abstract syntax, the concrete syntax and tree building function calls. We will illustrate the exploitation of Centaur by examples relying on CML concept statement. Let us recall its BNF definition [21]:

```
concept-def ::= concept Concept-name;
[terminology]
[sub-type-of: Concept-name <,Concept-name>*;]
[properties]
[axioms]
end concept [Concept-name;] .
```

First the programmer defines the abstract syntax corresponding to the statements to be parsed. The example below shows the Metal definition of the CML concept statement:

```
abstract syntax
concept  -> CNAME CON_BODY ;
con_body  -> DESCR SUPERC_LIST
             PROP_LIST AXIOMS ;
```

Then the programmer defines the language concrete syntax, thanks to syntax rules in a BNF like format:

```
<concept> ::=
"concept" <c_name> ";" <description> <superc>
            <properties> <axioms>
"end concept";
concept(<c_name>,
           con_body
              (<description>, <superc>,
               <properties>, <axioms>))

<superc> ::=
    "sub-type-of" ":" <superc_list> ";" ;
    <superc_list>

<superc_list> ::= <c_name> ;
    superc-list ((<c_name>))

<superc_list> ::=
    <c_name> "," <superc_list> ;
    superc-pre (<c_name>, <superc_list>)
```

Such rules are used by Centaur to create a parser for the language. This parser will build an abstract syntax tree for each parsed program of the language and the parsed program will be manipulated by means of its abstract syntax tree. All semantics manipulation will be stated as operations on abstract syntax trees. Centaur provides the support to handle such trees by means of specialized formalisms and manages the data structures underlying the trees. Thus, the programmer does not need to commit himself to an implementation for CML programs: the implementation is provided by Centaur, according to the language syntax specifications.

2.3 Definition of a pretty-printer for the language

In order to pretty-print abstract syntax trees in a readable format, the PPML formalism enables the programmer to specify pretty-printers. PPML is mainly a box language that allows to specify pattern-directed pretty-printing machines. The following example shows rules to pretty-print a tree associated to the CML concept statement:

```
concept(*name, con_body(*desc, *super,
           *prop_list, axioms)) ->
    [<v 4,0> [<h> "concept" *name ";"]
        *descr *super *prop_list *axioms
        "end concept"];

superc(*sup, **super_list) ->
   [<h> "sub-type-of :"
   *sup (<h 0> "," <h> **super_list) ";"];
```

Several pretty-printers can be written for the same tree and enable different but coherent presentations of this tree.

2.4 Definition of the language semantics

The language semantics is specified by the programmer by means of natural semantics rules [15], through the Typol formalism. Natural semantics, inspired of Plotkin's "structural operational semantics" [19], is a method allowing to define semantic specifications. As detailed in [15], a semantic definition is an unordered collection of rules. Intuitively, if all premises of the rule (i.e. the formulae of its numerator) hold, then its conclusion (i.e. the single formula of its denominator) holds. More formally, from proof-trees yielding the rule premises, a new proof-tree yielding the rule conclusion can be derived. Typol is an implementation of natural semantics. Typol rules are compiled into Prolog in order to be executed. Thus, once the formal semantics of the language is defined and compiled into Prolog by the Typol compiler, it is directly executable.

Below we show examples of Typol rules for the CML language. The first rule below expresses that in order to type-check a concept definition, Centaur must type-check first the super-type list (called Supers) and then the property list (called Properties). The next rule expresses how to type-check a super-type list by type-checking the first element and then the rest of the list. The last rule expresses that the type-checking of an empty list is satisfied.

```
DefList |- Supers & DefList |- Properties
-----------------------------------------

DefList |- concept(Name,
     con_body(Descr, Supers, Properties,
                       Axioms)) ;
```

```
definedconcept(DefList |- Sup => Concept)
& DefList |- SupList
--------------------------------------------

DefList |- superc[Sup.SupList] ;

DefList |- superc[] ;
```

2.5 Generation of Cokace by Centaur

We defined the whole abstract and concrete syntax of CML in Centaur, using Metal formalism. We used Typol formalism to define CML natural semantics, thus providing an operational semantics for CML. This work allowed to detect some ambiguities of early CML description, that were corrected in [21] and in KACTUS version of CML [23] : syntactic ambiguity of the relation statement, ontology used in inference knowledge, etc. Some ambiguities remain such as inheritance rules for concept subtyping. We wrote several pretty-printers in PPML formalism (one generating an HTML version of the CML program). Last, we also used Typol to implement a CML interpreter. Then Centaur generated Cokace, a CML-dedicated environment. The next sections will detail Cokace functions, its architecture and its interest for knowledge modelling.

3 COKACE OVERVIEW

3.1 CML

The BNF definition and the principles of CML are detailed in [21]. A CML program defines an expertise model made of domain knowledge, inference knowledge, task knowledge, and problem solving method knowledge. Domain knowledge can include ontology definition, ontology mapping definition, and domain model description. An ontology comprises a terminology (i.e. made of (a) descriptions of concepts, attributes, relations and expressions, (b) synonyms, (c) sources and (d) translation), definitions and can import other ontologies. A domain model uses an ontology and can comprise properties, expressions and annotations. Inference knowledge includes input roles, output roles, static roles, and the inference specifications. The inference level describes the elementary inference steps that can be conducted on domain knowledge. Task knowledge comprises task definition (i.e. goal, input and output of the task) and task body (i.e. task type, decomposition into subtasks, its control structure, and possibly assumptions). The task level describes the control level that directs inferences. Problem solving method knowledge describes the problem solving method inputs and outputs, competence specification, decomposition, control structure, and acceptance criteria.

3.2 Cokace architecture

The Cokace environment generated by Centaur is aimed at being used by a KE in order to build, verify and evaluate CML expertise models. As shown in figure 1, Cokace offers him the following support: (1) a structured editor, that helps the user to write syntactically correct CML programs, (2) a type-checker, that helps him to check the typing in CML programs, (3) and an interpreter, that helps him to run or simulate (parts of) CML programs.

3.3 Cokace structured editor

The Cokace structured editor for CML was generated by Centaur thanks to the CML syntactic definitions we provided. It supplies pattern-directed writing by means of generic statement patterns including so-called meta-variables. For example, if the user asks to create a new concept, the structured editor will introduce this pattern:

CML syntax				editor
CML static semantics	→	Centaur	→	type-checker
CML dynamic semantics				interpreter

Figure 1. Generation of Cokace through Centaur

```
concept CNAME1 ;
    TERMINOLOGY1
    sub-type-of : CNAME ;
    properties :
        PNAME1 : TYPE1 ;
        CARD1 DIFF1 DEFAULT1
    AXIOMS1
end concept
```

The user can then progressively construct his concept by instantiating the meta-variables appearing in this pattern: CNAME1, TERMINOLOGY1, etc. When the user clicks on such a meta-variable, the editor shows the possible generic patterns that may instantiate it. Then if the user clicks on one of the proposed patterns, the editor replaces the meta-variable by the pattern. The possible patterns that may be introduced depend on the current editing context and Cokace proposes only adequate patterns. For example, while the user edits the terminology part of an ontology, the editor proposes: concept, relation, expression and attribute patterns. The user can also visualize all the occurrences of a selected item, for example all occurrences of dynamic roles:

```
dyn_role
     initial_complaint -> complaint ;
     hypothesis -> state ;
     consistent_hypothesis -> state ;
     evidence -> set(manifestation) ;
```

He can then replace his current selection in the editor (e.g. here, a dynamic role) by one of the shown occurrences. He can also localize a selected occurrence in the CML program. The structured editor also offers structured cut, copy & paste editing.

As a conclusion, Cokace provides support for structured edition of CML syntactically correct programs. An interesting property of Centaur is that all these interactive structured editing tools were automatically generated, thanks to the specifications of the CML language.

3.4 Cokace type-checker

While the editor provides the user with syntactic help in order to write syntactically correct programs, the user also needs a semantics support in order to verify his expertise models. The type-checker offers such static semantics checking. It checks all that can be statically checked at definition time, in contrast with run time checking. Type-checking participates in early validating the expertise model while remaining at the CML level. It is possible to verify expertise at a conceptual level, before any implementation choice is made.

The type-checker operates on the abstract syntax tree of the CML program. Designed as a set of natural semantics inference rules, it is run by the Centaur logical inference engine. The type-checker verifies that all the references included in the CML expertise model only concern already defined entities. For example are the super-types of a concept defined? Are the referred attributes already defined? Are the concepts, attributes and relations referred in a domain model, defined in an ontology? The type-checker also performs inheritance checking. That is, if a referred attribute is not defined at the level of a concept, Cokace checks whether the attribute is inherited from its super-types. Domain model expressions are type-checked and the mapping between inference and domain layers is also checked. Signature mismatches are also checked on task and inference calls. Errors are detected and pointed out by means of active messages. When the user clicks on an error message, Cokace scrolls the editor containing the erroneous program on the line where the error occurs and highlights the guilty statement.

Thus, static type-checking ensures the internal coherence of the knowledge described in the CML expertise model.

3.5 Cokace interpreter

The Cokace environment also enables the KE to run (parts of) an expertise model in order to early simulate reasoning with the modelled expertise, in presence of the expert, for evaluation purposes. Such reasoning is performed at a conceptual level, without being committed to an implementation.

3.5.1 Inference engines

Domain models (e.g. causal model...) are sometimes stated by rules such as causal or behavioural rules. In such cases, it is possible to simulate reasoning by considering those rules as production rules. Therefore, we developed inference engines that operate on such rules by backward and forward chaining. The user selects the appropriate inference engine by means of the operation-type keyword associated to the inference. For example, in the following inference, the KE exploits a backward inference engine to generate all the hypotheses relevant to explain a fault.

```
inference cover
    operation-type : backward ;
    input-roles :
       initial_complaint -> complaint ;
    output-roles : hypothesis -> set(state) ;
    static-roles :
       potential_cause -> cause
           in causal_model ;
```

3.5.2 Control language

As the standard CML task control structure consisted only of text, we extended it by proposing a control language over inference and task calls, in the spirit of the examples given in [21]. This control language provides variable passing and control operator such as sequence, conditional and loop. The control language also offers recursive task calls. The interest of this control language is that it enables a deeper verification and evaluation through the simulation of the task.

3.5.3 Simulating tasks or inferences

The interesting point is that the interpreter can not only run a task but also directly run an inference when given correct inputs. So, it is possible to conduct computer-aided expertise model validation, as the user can test several inferences over specific domain models. For example, the user can ask Cokace to check all the consequences of a fault, to compute the possible diagnoses of a complaint, etc. The user can launch a reasoning by means of such a request as: cover(engine_does_not_start = true) where cover is the name of an inference (resp. a task) and engine_does_not_start = true is a correct input for this inference (resp. task). The following example shows a test diagnosis task:

```
task test ;
task-definition
  goal : "test a potential solution";
  input :
    hypothesis : "potential solution";
    data : "observable data";
  output : hypothesis: "solution";
task-body
  type : primitive ;
  sub-tasks : establish ;
  additional-roles : consistent_hypothesis;
  control-structure :
   test(h:hypothesis, d:data -> h:hypothesis)
   = establish(h:hypothesis, d:data ->
               h:consistent_hypothesis);
```

Its simulation, with a hypothesis and user data given as input parameters, allows to verify that its control structure is satisfied: adequate call to the establish inference, with parameters respecting the input and output roles specified in this inference. In conclusion, the KE and the expert can check and evaluate expertise at the knowledge level by simulating CML expertise models.

4 DISCUSSION

In comparison with the tools dedicated to the KADS method such as KADS Tool [10], or Open-KADS [6], our originality is (1) the exploitation of the Centaur system and of all its syntactic and semantic verification abilities, (2) our relying on the CML language.

With respect to related research on validation during knowledge acquisition phase, the possibility to simulate CML expertise models directly, seems rather original. It can be compared to the software support offered by the TheME tool for formalizing an expertise model [3], for the construction and validation of a formal model. In Si$(ML)^2$ [27], an interpreter for a subset of $(ML)^2$ allows an interactive simulation. CML is generally considered as an informal language (a "semi-formal language", according to its designers): it allows to describe informal but highly structured expertise models. Its informal feature may be considered as making easier communication between the KE and the expert. But the need to avoid informal language ambiguities and to operationalize CML expertise models has lead researchers to propose a translation of CML into a formal language such as $(ML)^2$ [1, 29]. Guidelines help to build an $(ML)^2$ formal specification from an informal CML expertise model [2, 1]. In our approach, the KE needs not build a formal model from an informal CML model: he works on his CML expertise models and benefits from verification capabilities directly on this model. Moreover, he has no specific programming work: we already did this work by describing CML syntax and semantics in Centaur and by writing adequate inference engines.[2]

[2] Our work on operational semantics of CML and on the control language can be considered as a way to formalize CML.

Then, all verification abilities are automatically provided by Centaur. However, the general guidelines stressed in [1] are still useful, of course.

The system currently implements all CML but the following statements : structure, import in an ontology, ontology mapping and problem solving methods. The type-check is complete and interpretation is possible with rule based domain models.

Our work shows an example of the benefits knowledge engineering can get from SE technologies: SE tools can help to design/develop knowledge engineering environments. Such link between knowledge engineering and SE is more and more emphasized [14], in particular in research on formal methods for knowledge engineering: $(ML)^2$, KARL [8], DESIRE [16], KBSSF [13], CERISE [30].

5 CONCLUSION

The Cokace prototype shows how an advanced SE environment such as Centaur can be exploited to design and build an environment for expertise modelling within the CommonKADS framework. We thus developed a set of tools dedicated to CML language: a structured editor, a type-checker and an interpreter. The resulting environment helps the KE to design and early validate/verify CML expertise models without being committed to implementation choices: he can remain at the knowledge level.

An application is currently under progress in Cokace on medical expertise in breast cancer prognosis and therapy: Cokace was already tested on a simple example of CML program in this field and at present, a more complete CML expertise model in this domain is being developed using Cokace [20]. Experiments will also be carried out in order to implement parts of the CommonKADS generic modelling library within Cokace. It will enable to check the correctness of the CML descriptions of the CommonKADS library generic models [5]. We will implement and test more complex inference engines. Centaur can also be used to compile CML into a target language (such as Prolog), or into a knowledge representation formalism. We currently exploit this feature, in order to operationalize CML expertise models into Sowa's conceptual graphs [24], so that a conceptual-graph-based tool can manage the knowledge thus represented at the symbol level.

ACKNOWLEDGEMENTS

We deeply thank the Croap team at the INRIA that kindly provided us with invaluable help, and especially Yves Bertot, Thierry Despeyroux, Francis Montagnac, Valérie Pascual and Laurence Rideau.

REFERENCES

[1] M. Aben, *Formal Methods in Knowledge Engineering*, Phd thesis, Univ. of Amsterdam, Feb. 1995.

[2] M. Aben, J. Balder, and F. van Harmelen, 'Support for the Formalization and Validation of KADS Expertise Models', ESPRIT Project P5218 KADS-II Report KADS-II/M2/UvA/DM2.6a/1.0, Univ. of Amsterdam, (1994).

[3] J. R. Balder and J. M. Akkermans, 'TheME: an environment for building formal KADS-II models of expertise', *AI Comm.*, **5**(3), 136–147, (1992).

[4] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, 'Centaur: the system', in *Proc. of the 3rd Symp. on Softw. Dev. Env, ACM SIGSOFT'88*, pp. 14–24, Boston, MA, (Nov 1988).

[5] J. Breuker and W. Van de Velde, *CommonKADS Library for Expertise Modelling, Reusable Problem Solving Components*, IOS Press, Amsterdam, 1994.

[6] Bull, 'Open-KADS Documentation', Technical report, (1992).

[7] D. Canamero and S. Gedolf, 'Coupling Modelling and Validation in COMMET', in *Proc. of EUROVAV'93*, Mallorca, Spain, (June 1993).

[8] D. Fensel, J. Angele, and D. Landes, 'KARL, a Knowledge Acquisition and Representation Language', in *Proc. of the 11th Int. Conf. on Expert Syst. and their Appl*, pp. 513–528, Avignon, (May 1991).

[9] S. Gedolf and A. Slodzian, 'From Verification to Modelling Guidelines', in *A Future for Knowledge Acquisition : Proc. of EKAW'94, LNAI n. 867*, pp. 226–243, Belgium, (Sept. 1994). Springer-Verlag.

[10] Ilog, 'KADS Tool: Methodological Guide: Version 1.0', Tech. rep., (1993).

[11] ISL, 'The CommonKADS Workbench', Tech. rep., (1996).

[12] I. Jacobs and L. Rideau, 'A Centaur Tutorial', Tech. Rep. 140, INRIA, (1992).

[13] W. Jonker and J. W. Spee, 'Yet another formalization of KADS Conceptual Models', in *Proc. of EKAW'92*, ed., T. Wetter et. al., pp. 211–229. Springer-Verlag, (May 1992).

[14] W. Jonker, J. W. Spee, I. Vled, and M. Koopman, 'Formal approaches towards design in Software Engineering and their roles in KBS design', in *Proc. of IJCAI'91 Work. on Softw. Eng. for KBS*, pp. 84–98, Sydney, Australia, (Aug. 1991).

[15] G. Kahn, 'Natural Semantics', in *Proc. of the 4th Annual Symp. on Theor. Aspects of Computer Science, STACS'87, LNCS n. 247*, pp. 22–39, Passau, (1987). Springer-Verlag.

[16] I.-A. Langevelde, A.W. Van Philipsen, and J. Treur, 'Formal Specification of Compositional Architectures', in *Proc. of ECAI'92*, pp. 272–276, Vienna, Austria, (Aug. 1992).

[17] J.-P. Laurent, 'Proposals for a valid terminology in KBS Validation', in *Proc. of ECAI'92*, ed., B. Neumann, pp. 829–834, Vienna, Austria, (Aug. 1992). John Wiley & Sons, Ltd.

[18] A. Newell, 'The Knowledge Level', *Artificial Intelligence*, **18**(2), 87–127, (1982).

[19] G. D. Plotkin, 'A Structural Approach to Operational Semantics', tech. rep. DAIMI FN-19, Computer Science Dept, Aarhus Univ., (Sept. 1981).

[20] R. Sacile, 'Using CommonKADS to build an expertise model for breast cancer prognosis and therapy', INRIA Research Report 2737, (1995).

[21] G. Schreiber, B. Wielinga H. Akkermans W. van de Velde, and A. Anjewierden, 'CML: The CommonKADS Conceptual Modelling Language', in *A Future for Knowledge Acquisition: Proc. of EKAW'94*, ed., L. Steels et. al., pp. 1–25. Springer-Verlag, (Sept. 1994).

[22] G. Schreiber, B. Wielinga, R. de Hoog, H. Akkermans, and W. Van de Velde, 'CommonKADS: a Comprehensive Methodology for KBS Development', *IEEE Expert*, **6**(9), 28–37, (Dec. 1994).

[23] G. Schreiber, B. Wielinga, and W. Jansweijer, 'The KACTUS View on the 'O' Word', in *Ontology Dev. and Use Workshop*, (1994).

[24] J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Reading, MA. Addison-Wesley, 1984.

[25] L. Steels, *COMMET: a componential methodology for knowledge engineering*, 1991. Esprit Project CONSTRUCT DWP/2/3/4.

[26] L. Steels, 'The componential framework and its role in reusability', in *Second Generation Expert Systems*, ed., J.-M. David et. al, pp. 273–298, Berlin, (1993).

[27] A. Ten Teije, F. van Harmelen, and M. Reiders, '$Si(ML)^2$: a prototype interpreter for a subset of $(ML)^2$', ESPRIT Project P5218 KADS-II Report KADS-II/T1.2/TR//UvA/005/1.0, Univ. of Amsterdam, (1991).

[28] P.-A. Tourtier and S. Boyera, 'Validating at Early Stages with a Causal Simulation Tool', in *Proc. 8th European Knowledge Acquisition. Workshop, EKAW'94, LNAI 867*, eds., L. Steels and W. Van de Velde, pp. 303–317. Springer-Verlag, (1994).

[29] F. van Harmelen and J. R. Balder, '$(ML)^2$: a formal language for KADS models of expertise', *Knowledge Acquisition, Special issue on "The KADS approach to knowledge engineering"*, **4**(1), 127–161, (1992).

[30] C. Vicat, J.-G. Ganascia, and A. Busac, 'Modularity in knowledge acquisition: a step towards reusability', in *Proc. of KAW'95*, pp. 36.1–36.17, Banff, Canada, (Feb-March 1995).

[31] B. Wielinga, W. Van de Velde, G. Schreiber, and H. Akkermans, 'Expertise Model Definition Document', ESPRIT Project P5218 KADS-II Report KADS-II/M2/UvA/026/1.1, University of Amsterdam, (1993).