Knowledge Representation for Program Reuse

Sabine Moisan

Knowledge Representation for Program Reuse

Sabine Moisan[1]

Abstract. In recent years, programs and knowledge about programs have become an important part of the "patrimony" (or intellectual property) of companies. However, the scope of this knowledge is wider than the simple code sources. In order to manage this knowledge it is necessary to model it. We propose a novel framework to encompass such knowledge. In this framework, we have defined a general ontology for program management and a knowledge description language, for documenting, modelling, and capitalising on the knowledge about the use of code. The paper presents the concepts of the ontology, their concrete representation in the language and some utilisations by different inference mechanisms.

1 INTRODUCTION

The use of computer programs has become a common practice in many activities. A large number of programs have been developed by specialists in one discipline, yet often used by non-specialists for varied application purposes. Since programs are more and more complex, their documentation, their maintenance and their evolution have become a major concern. Programs are often viewed as limited to source or object codes. But an effective long term management should also take into account attached knowledge as diverse as knowledge about the purpose, the scientific foundations, the intended applications, the conditions of applicability, the results of past tests, the know-how of everyday end users, etc. This versatile knowledge has been accumulated over the years (the life-time of the programs) and is scattered among different people, who are "sources" of knowledge about programs, either because they know the theory behind the code, because they have run the programs on numerous data, or because they have written or modified the code. Moreover, not only individual programs but also useful combinations of several programs to perform complex tasks are part of the knowledge.

Indeed, the programs and their use belong to a company "patrimony" that should not be lost and that should be easy to re-use and to maintain. That is why companies need to keep track of all the necessary skills for the optimal use of programs, for both user assistance and knowledge management purposes. As an answer to this issue, we propose an approach based on:

- A conceptual model for experts, designers and users of programs allowing them to communicate about programs and their use with a unified terminology (based on generic concepts, such as data, programs, sequences of programs, data flow, etc.). Such concepts are recurrent and can be gathered in a *general ontology*;
- A descriptive *language* to represent and manipulate abstract concepts. For this purpose, we have defined YAKL, an open language which provides experts with a user-friendly syntax and a well defined semantics for the concepts in our model;
- Computer *tools* to ensure the consistency of the expressed knowledge, to operationalise it into computer data structures and to produce effective systems to help run them (semi)automatically, provided that all the necessary knowledge is properly formalised

In this paper we focus on the model and the way it is operationalised in the YAKL description language in order to document, model and capitalise all knowledge about the use of codes.

2 KNOWLEDGE AND PROGRAMS

The management of programs is generally performed by a person (termed the *expert* in the following) and it relies on a large amount of knowledge. Not only the code lines, but also the knowledge about how to run programs, how to evaluate their results, how to tune them, how to combine them for higher level computations, etc. is necessary. Thus, when experts -who have this know-how- are not available or when they retire or leave, it is necessary to keep this processing knowledge in an understandable and possibly operational form. Such knowledge is seldom made explicit in documentation and cannot be found in source codes.

Our objective is to model only what is relevant to communicate about programs and to manipulate (and run) them, without exposing their code. For this purpose, we define *representations* of programs and we provide *composition operations*, such as sequence or alternative, to produce higher-level combinations for complex tasks. In the following, we use the general term of *operator* for the representations of both real programs and combinations.

Modeling the knowledge required to obtain optimal performance from programs can be viewed as "encapsulating" programs, adding layers of different kind of knowledge (*syntactic, strategic* and *semantic*) to source codes. Syntactic knowledge consists of calling syntax, order and type of input/output arguments, or even information such as operating system or memory required. Strategic knowledge corresponds to the way to assemble programs for complex tasks. Semantic knowledge is the specialist's know-how about the use of the programs and the decisions that should be made: e.g. what are the discriminant characteristics of a program, how to perform result evaluation or failure handling. Such packaging enhances the program with all the necessary knowledge to use and re-use it in different situations, to document it and to help maintain it. The result is understandable and reusable by other people in addition to the specialist who designed or implemented the code.

More formally, our model is based on a set of argument types and a set of operators. For a particular application, Ω denotes the set of available operators and Υ the set of their input and output argument types. Each operator $\omega \in \Omega$ is represented by: $(\mathcal{I}_\omega, \mathcal{O}_\omega, \mathcal{P}_\omega)$, where $\mathcal{I}_\omega$ (resp. $\mathcal{O}_\omega$) $\subset \Upsilon$ is the ordered set of types of ω input (resp. output) arguments and $\mathcal{P}_\omega$ the "protocol of use" related to ω (i.e. the semantic knowledge, in the form of a set of inference decisions

[1] INRIA, BP 93, 06902 Sophia Antipolis, FRANCE

to manipulate the operator). For the strategic knowledge, the model provides several composition operations: sequence, alternative, parallel, iteration, etc. to recursively organise operators into more abstract ones.

3 ELEMENTS OF ONTOLOGY AND LANGUAGE

To achieve our modeling goal, we first define an ontology which contains general concepts, such as data or programs. This ontology provides experts with "patterns" (or reusable templates) that they will instantiate with respect to a domain[2], thereby obtaining a domain ontology. For example, if image processing is the domain, they may obtain an image-processing ontology containing the description of images and image processing programs. Refining a step further, they may even focus on a particular application, such as flaw detection; in this case they obtain an application ontology, *e.g.*, containing the definition of artefact images and specific flaw detection programs.

Second, we design the YAKL language that offers a concrete syntax to describe the concepts of the abstract model, at the right level for each development role. It provides programmers with a way to document their programs, technicians with a way to note guidelines to appropriately use programs, scientists with a way to annotate programs and to link them with formulae or theoretical papers (and vice versa i.e. to find programs connected with the same theory). YAKL is a means to describe the knowledge about a set of programs, independently of any implementation language, any domain, or any application. It is used both as a common storage format for knowledge and as a human readable format for writing, exchanging, and consulting knowledge. We have in parallel defined a formal semantics for the language. From an operational point of view, the language can also be translated into computer structures to produce an executable knowledge-based system. YAKL already captures most knowledge about program use, even elements which are seldom explicited in other approaches (e.g., repair strategy). Furthermore it is an *open* language that can be extended or adapted to suit different needs.

It should be noted that the language provides a syntax uniquely for the general ontology (referring to common concepts, such as "program", "argument", etc.). Based on it, experts can build knowledge bases to define and use other kinds of ontologies (domain and application ontologies mentioned before), which are out of the scope of YAKL (e.g., an image processing ontology, referring to image processing concepts).

We have identified the concepts that play a role in program use and modeled them in order to get the most widely usable representation. As a result we propose guidelines that enable the representation of programs and issues that play a role in the composition of a solution using the programs. It is thus also a guide on how to (re-)use them. The terminology we have chosen for the concepts is the result of an analysis [19, 4] of many existing systems related to program management that we have either developed or closely studied. Even if each system has its own vocabulary, some terms (like "operator") are widely used. The next sections define the concepts of the proposed general ontology and their concrete representation in YAKL. The main concepts detailed are the *operators*, with their *arguments* and attached *criteria*. YAKL uses both frame-based and rule-oriented descriptions. Frames are used for operators or arguments, whereas inference rules are used for criteria.

[2] An application domain refers to the object (focus of cognition) of the programs, for instance mathematics or image processing are possible application domains.

3.1 Operators and Arguments

Operators represent either concrete programs (primitive operators) or abstract processing (composite ones). Both have input and output arguments and encapsulate various criteria in order to manage their input parameter values (initialisation criteria), to assess the degree of quality of their results (evaluation criteria on output data), or to react in case of poor results (repair criteria). Several operators (of both types) may have to be applied to achieve a single user's abstract processing.

The common operator representation uses the frame formalism and includes (most items are optional):

- A reference to an abstract functionality, *i.e.*, processing objective (*e.g.*, "thresholding" or "segmentation" may be defined as functionalities in image processing).
- Characteristics: a symbol list describing non functional characteristics of an operator, known by the expert (*e.g.* "slow, resource_consuming").
- Information on arguments, including their names, types, ranges or means to compute their value (expresses by slot facets).
- Pre- and post- conditions on in/output arguments, to be checked before and after the execution of an operator.
- Expected effects, to describe what the operator achieves and what its effects are on the outputs.
- Various criteria, to specify the reasoning made on operators (in the form of rule bases, see 3.2).

Arguments are represented as operator slots. They play an important role because many decisions (e.g., the selection of a program) are based on the information that arguments provide. This is particularly true if processing is data-driven, as in image processing. Some usual types, such as integer or float, are predefined in the Υ set of types, but most of them are defined by experts. YAKL provides them with a frame-based representation and a hierarchical organisation for argument types. The model differentiates two classes of arguments: data and parameters. Data have fixed values which are assigned for input data (*e.g.*, an input raw pixel image), or computed during the reasoning process, for output data (*e.g.* an output segmented image). Parameters are tuneable, *i.e.* their values can be set by means of initialisation criteria or modified by means of repair criteria. Parameters are always input arguments. The values of output data can be "assessed" by means of evaluation criteria, and these judgements drive the process of result evaluation and parameter adjustment.

For instance the YAKL source to define a new argument type (`Polynom`) for mathematical processing is defined below, simply as an extension of a file (containing the text of a polynomial system, plus slots containing information about numbers of variables and of equations). This type will be added to Υ and it can be used latter to type operator arguments. YAKL syntax is close to natural expression, but more structured (keywords are indicated in bold face). In particular, the frame slots have several optional predefined facets (e.g. *default* or *range*).

Argument Type { **name** Polynom **Subtype Of** File
  Attributes
  Integer **name** nb_variables
    default 1
  Integer **name** nb_equations
    default 2 }

Operators representing concrete programs are referred to as *primitive operators*. They describe the programs as "black boxes" known only by information on how they can be used in different situations and by their input and output arguments. In addition to the common

information, their descriptions contain the information needed for effective execution of programs (including calling syntax). The execution of a primitive operator corresponds to the execution of its associated program.

The structural part of YAKL code to describe an image thresholding operator is given below (we suppose that a type `Image` has been previously defined, with a `noise` attribute). It details the achieved functionality (thresholding), input and output arguments, a precondition on image noise, and the calling syntax, which has to be instantiated at execution time with the actual values of arguments.

Primitive { **name** thresh
Functionality thresholding
Input Data
Image **name** image1 **comment** "original image"
Input Parameters
Float **name** threshold
default 1
Output Data
Image **name** image2 **comment** "thresholded image"
....
Preconditions image1.noise == gaussian
(Criteria omitted ... see 3.2)
Call
language shell
syntax cd image1.path ";" thresh -s threshold image2 }

Composite operators represent higher level operations. They break down into more and more concrete (composite or primitive) sub-operators. They therefore correspond to useful decompositions that are predefined by the expert. These decompositions at different levels of abstraction must end with primitive operators. Currently, we offer alternative (|), sequence (-), parallel (||), and iterative ($*$) as types of decomposition. In a sequential decomposition some sub-operators may be optional. Alternative decompositions provide a way of grouping operators into semantic groups corresponding to the common functionality they achieve. This is a natural way of expression for many experts because it allows levels of abstraction above the level of programs. In addition to the common information, the way to refine a composite operator is expressed by:

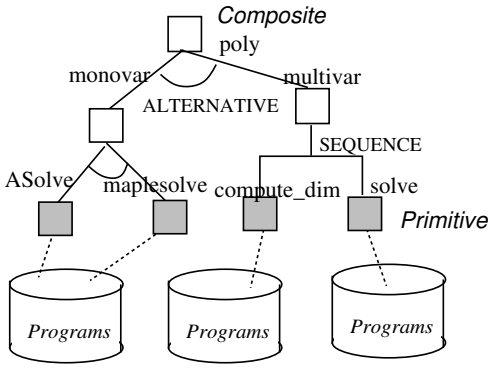

Figure 1. Mathematical operators. Composite ones are represented by white squares, primitive ones by grey squares.

- Control information about the type of decomposition into sub-operators,
- References to sub-operators,
- Data flow information between a parent operator and its children (and between children operators in a sequential decomposition),
- Additional criteria (for choices, optional applications of sub-operators in a sequence, and repair strategy).

The YAKL frame corresponding to the composite operator `multivar` of figure 1 is presented below. It uses the previously defined argument type `Polynom`.

Composite Operator { **name** multivar
comment "solve polynomial systems with nb. variables >1"
Input Data Polynom **name** Sy
Output Data Polynom **name** sol
Preconditions Sy.nb_variables > 1
Body compute_dim **-** solver *(- stands for a sequence)*
Distribution *(data flow parent-children)*
multivar.Sy **/** compute_dim.PSin
multivar.sol **/** solve.sol
Flow compute_dim.PSout **/** solve.Sy *(data flow among children)* }

3.2 Criteria

Different types of criteria can be attached to operators: common criteria attached to both composite and primitive ones and additional criteria for composite operators only. Criteria represent the dynamic knowledge about *decisions* (e.g., how to choose among alternatives or how to adjust the processing with the determination of new input values for programs or the selection of other programs). Criteria provide a system with flexible reasoning facilities. For the time being, the criteria are represented in YAKL by specialised rule bases (groups of rules) which are attached to operators. Initialisation of parameters, result evaluation, repair and adjustment rule bases can be attached to all operators, while choice and optional criteria are specific to composite ones. The locality of the criteria allows each piece of knowledge to carry its own decision knowledge with respect to its role in the processing and the kind of information it has access to.

The following table summarises the main types of criteria, in the form of abstract inference rules with their typical conditions and actions (other types of conditions and actions are possible). Several rules of each type constitute a criterion expressing the expert's know-how on a particular reasoning decision.

Choice	Initialisation
If Object attribute *a* has value *v* **Then** Use program *p*	**If** Object attribute *a* has value *v* **Then** Set parameter *p* to value *v1*
Assessment	Repair
If Result *r* has property *p* **Then** Declare problem *pb* for *o2* Declare problem *pb* for *o2*	**If** Operator *o1* has problem *pb* **Then** Transmit *pb* to operator *o2*

As an example, below is the choice criteria of operator `poly`, that decides whether to choose `multivar` or its alternative `monovar`:

Choice criteria
Rule { **name** choice_mono
If PS.nb_variables **==** 1
Then use_operator monovar }
Rule { **name** choice_multi
If PS.nb_variables > 1
Then use_operator multivar}

3.3 Interrelations of Concepts

The ontology not only defines the concepts but also their relationships, which are summarised in figure 2. It is an abstract, simplified

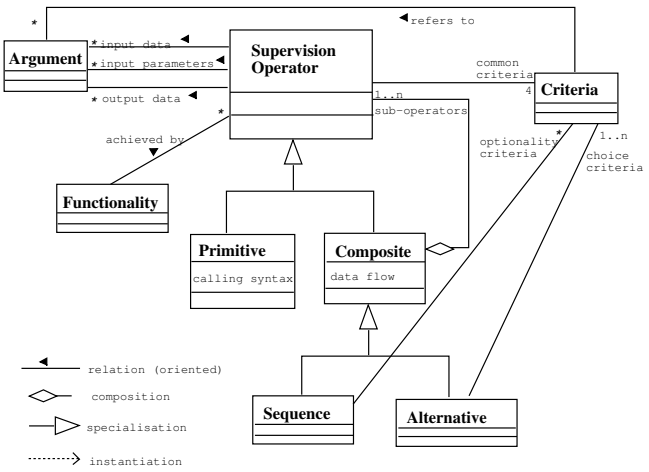

Figure 2. Relations between concepts in the ontology.

view in the form of an UML (Unified Modelling Language) class diagram. Most of the associations are "one to many", *e.g.*, one operator is connected to many inputs (in UML notation, * denotes 0 or more) and to 0 up to 4 common criteria: since criteria are not compulsory, an operator may have no criteria at all. In the same way, a composite operator is connected by an aggregation relation (denoted by a diamond) to several (at least one) sub-operators. Not all the possible types of composite operators are represented in this simplified view, only sequence and alternative types, since they are the most commonly used. A primitive operator is a leaf of the hierarchy. A functionality may be achieved by several operators.

4 USE OF THE LANGUAGE

All concepts of our ontology are to be managed by inference mechanisms relying on knowledge description. YAKL provides experts with general framework to store their knowledge about programs in a knowledge base. Such a knowledge base contains only information that plays a role in program management, i.e. descriptions of operators, their arguments, their competence and applicability conditions of operators, their relations, etc.

The YAKL source of a knowledge base can be parsed and eventually translated into various formats to be processed by computer tools: an inference engine for execution, a graphical tool for visualisation, a simulator, etc. During the parsing, syntactic and semantic verifications are performed on the knowledge description: e.g., type checking in assignments, type compatibility between argument value type and default value or range, warning if parameters have no initialisation method (default value or initialisation criteria), etc.

A knowledge base designed with YAKL can potentially be run by several inference mechanisms, provided that it contains the required information for the inferences. Different inference mechanisms may not use the same knowledge parts or not in the same way.

For instance, we have implemented different knowledge-based system engines to exploit knowledge on programs in order to produce a plan of programs that achieves a user's processing goal. We call this activity *program supervision* (PS), the objective is to mimic the strategy of an expert in the use of programs, to explore the different possibilities and to compute the best one, with respect to available concepts descriptions. In parallel with the specification of the general ontology described above, we have developed a general problem-solving method for PS. It, which includes all the data processing steps:

1. Problem identification in term of a functionality to achieve,
2. Construction of a proposal (selection and rank-ordering of programs, based on composite operators, arguments, pre/postconditions and effects),
3. Effective execution of codes (based on primitive operator descriptions),
4. Evaluation of result quality (by evaluation criteria),
5. Repair in case of problems (by repair and adjustment criteria, to reorder the proposal or to reexecute current operator after modification of its parameter values).

Each phase relies on the semantics of the knowledge it uses. For example, in phase 3, a prerequisite to executing a program is to initialise the values of its parameters: it is the role of the initialisation criteria in our model. We have defined a denotational semantics for all concepts in the general ontologybut its description is out of the scope of this paper.

The main basis of the general ontology was the experience with OCAPI [3]. The design of three new engines led to *variants* [4] of the different phases and thus implied modifications of the ontology and of the syntax of the description language. These modifications are briefly described hereafter.

The PEGASE engine refines the hierarchical planning method of OCAPI. In particular, it introduces the concept of *optional* sub-operator in a sequential decomposition and the corresponding new kind of expert-defined criteria. Another important improvement concerns the failure handling mechanism [13], which introduces another kind of criteria (*Repair*).

The PULSAR [18] engine combines hierarchical and dynamic operator-based planning methods. This second planning method matches the description of both the type and the contents of inputs and outputs with operator preconditions and effects. These concepts are thus better exploited than in PEGASE. In addition, PULSAR introduces *unordered decomposition*, a new type of composite operator decomposition and *weights* for attributes of argument types.

Finally, the MEDIA engine introduces additional concepts needed for its hybrid and perspective-based planning method, for example *weak preconditions* on operators (preconditions that allow a better fit with data to be analysed and with objective, but can be relaxed when an optimal solution cannot be reached).

In order to adapt and extend the ontology (and the language accordingly), we propose a generic and customisable software development platform, devoted both to knowledge base and inference engine design. This platform thus integrates ontological as well as problem-solving models. The task ontology corresponds to templates for knowledge base contents; it is implemented as a library of reusable components (abstract classes) that can be derived when ontology extensions are needed. A knowledge base editor that supports YAKL is also provided by the platform. It is parametrised by the grammar of the language. An evolution of the syntax thus corresponds to a change in the grammar rules. Such an approach allows to reuse existing elements when possible, to extend them when necessary or to consistently add new ones without modifying the others.

For instance, the ontology extensions for the three engines led to the following existing class derivations: to accommodate the concept of optional sub-operators, PEGASE introduced new sub-classes of *Link* between operators and of *Criteria*; PULSAR derived class *Decomposition* to express unordered decomposition type and class *Argument* to introduce weights, and in MEDIA, a derivation of the

Condition class implements the "weak condition" concept. The syntax of YAKL version for each engine has been modified too, by introducing new keywords and/or new rules: e.g., new *Optionality Criteria* keyword for PEGASE, or a new grammar rule to accept weak preconditions (inside square brackets) for MEDIA.

Based on the same approach, we are currently working on distributed program supervision or program brokering on the Web, where YAKL and its semantics are used to search for programs to achieve a distant user's goal.

5 DISCUSSION AND CONCLUSION

Unlike other related solutions for program management [1, 8, 9, 10, 2], which are often motivated by application issues and hence are committed to their domains, our approach provides an abstract and generic point of view on program management activity. Moreover, in order to support extensions, we have applied a strict "separation of concerns" policy at every level, from the ontology definition to the computer tools in the platform. Thus both the ontology and the YAKL syntax are *extensible* and can be refined and adapted e.g., to different target domains or purposes. Current versions have proved sufficient to express the necessary knowledge in most cases studied so far [16], but our approach already enabled us to extend them for particular purposes (*e.g.*, resource management [12]).

Knowledge modeling work such as that presented in this paper yields a high level and intuitive explanation of program management problems, a major concern in today's industry. Furthermore, the approach provides experts with guidance for knowledge representation. The ontology helps make explicit the role of knowledge elements in PS (such as parameters) and allows them to identify missing or irrelevant knowledge (for instance lack of argument setting). The YAKL language has been designed to offer a model-based view of the use of programs which is easy to comprehend because it conceals implementation or domain-dependent details. Using this language, experts can express their knowledge at the expertise level, guided by dedicated representation patterns provided by the underlying ontology.

Several general purpose languages for knowledge or ontology definition and exchange have been developed (e.g., KIF [7], or more recently, OIL [6] for Web applications). Though YAKL also exhibits general knowledge modeling features, its major contribution lies in a natural description of strategic and semantic knowledge about software components (programs in our case), in a domain-independent way. The aim is to reason *about* these components, in the same line as recent work, e.g., on UPML [5] to describe problem-solving methods and to facilitate their reuse in an internet-based environment.

Other work on tasks and methods [17, 15] also identifies the need for adapted modeling languages. For instance, the AROM [14] language extension for tasks is based on concepts similar to ours. Interest in formalisms for reusing and reasoning about programs has also emerged in domains such as scientific workflow management [11].

YAKL encompasses the descriptive power of most of these languages (except for distributed features) and can be *adapted* to various needs. Its human-readable form is easily adopted by non computer scientists, yet it can also be translated into various formats (e.g. RDFS) in order to facilitate its interoperability with existing tools (e.g. on the Web). It helps *formalise* the description by providing a common language to experts, which is understandable across domains, with a formal semantics. Thus it enables *sharing* of knowledge between experts. Moreover, it can be used in an incremental manner: from simple code documentation to a real knowledge base for a program supervision system.

REFERENCES

[1] R. Bodington, 'A Software Environment for the Automatic Configuration of Inspections Systems', in *Proceedings of KBUP'95*, pp. 99 – 108. Sophia Antipolis, France, INRIA, (November 1995).

[2] S. A. Chien and H. B. Mortensen, 'Automating image processing for scientific data analysis of a large image database', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **18**(8), 854–859, (August 1996).

[3] V. Clément and M. Thonnat, 'A Knowledge-based Approach to the Integration of Image Processing Procedures', *Computer Vision, Graphics and Image*, **57**(2), 166–184, (March 1993).

[4] M. Crubezy, M. Marcos, and S. Moisan, 'Experiments in Building Program Supervision Engines from Reusable Components', in *3th European Conference on Artificial Intelligence Workshop on Applications of Ontologies and Problem-Solving Methods.*, (August 1998).

[5] D. Fensel, V. R Benjamins, E. Motta, and B. Wielinga, 'UPML: A Framework for Knowledge System Reuse.', in *International Joint Conference on AI (IJCAI-99)*, (July 1999).

[6] D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein, 'OIL in a Nutshell', in *EKAW'2000, 12th International Conference on Knowledge Engineering and Knowledge Management*, ed., R. Dieng and O. Corby, Lecture Notes in Artificial Intelligence 1935, Juan les Pins, France, (October 2000). Springer-Verlag.

[7] M. Genesereth and R. Fikes, 'Knowledge interchange format version 3.0 reference manual', Technical Report 94305, Computer Science Department Stanford University, Stanford, California, (1992).

[8] L. Gong and C. A. Kulikowski, 'Composition of Image Analysis Processes Through Object-Centered Hierarchical Planning', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **17**(10), (October 1995).

[9] A. Lansky, S. Friedman, L. Getoor, S. Shmidler, and N. Short, 'The COLLAGE/KHOROS Link: Planning for Image Processing Tasks', in *AAAI Spring Symposium on Integrated Planning Applications*, (March 1995).

[10] C.-E. Liedtke, H. Munkel, and U. Rost, 'Solution for a learning configuration system for image processing', in *11th International Conference on Industrial Applications of Artificial Intelligence and Expert Systems*, eds., A.P. del Pobil J.Mira and M. Ali, Lecture Notes in Artificial Intelligence 1415, Benicassim, Spain, (June 1998). Springer-Verlag.

[11] F. Llirbat, E. Simon, R. Hull, B. Kumar, J. Su, G. Zhou, and G. Dong, 'Declarative Workflows that Support Easy Modification and Dynamic Browsing', in *International Joint Conference on Work Activities Coordination and Collaboration*, pp. 69–78, (1999).

[12] S. Moisan and M. Thonnat, 'Program Supervision under Resource Constraints', in *3th European Conference on Artificial Intelligence Workshop on Monitoring and Control of Real-Time Intelligent Systems.*, (August 1998).

[13] S. Moisan, R. Vincent, J. van den Elst, , and F. van Harmelen, 'Towards an Intelligent Failure Handling Mechanism in Program Supervision', in *Proceedings of KBUP'95*, ed., INRIA, Sophia Antipolis, France, (November 1995).

[14] M. Page, J. Gensel, C. Capponi, C. Bruley, P. Genoud, and D. Ziébelin, 'Representation de connaissances au moyen de classes et d'associations : le système AROM', *Langages et Modèles Objets (LMO'00)*, 91–106, (janvier 2000).

[15] X. Talon and C. Pierret-Golbreich, 'TASK: a framework for the different steps of a KBS construction', in *Workshop on Knowledge Engineering and Modelling LAnguage (KEML'96)*, (1996).

[16] M. Thonnat, S. Moisan, and M. Crubézy, 'Experience in Integrating Image Processing Programs', in *Internatianal Conference on Computer Vision Systems*, Las Palmas, Canary Islands, (January 1999).

[17] F. Trichet and P. Tchounikine, 'DSTM: a Framework to Operationalize and Refine a Problem-Solving Method modeled in terms of Tasks and Methods', *International Journal of Expert Systems With Applications*, **16**(2), 105–120, (February 1999).

[18] J. van den Elst, *Modélisation de Connaissances pour le Pilotage de Programmes de Traitement d'Images*, Thèse, Univ. Nice-Sophia Antipolis, octobre 1996.

[19] J. van den Elst, F. van Harmelen, and M. Thonnat, 'Modelling Software Components for Reuse', in *Seventh International Conference on Software Engineering and Knowledge Engineering*, pp. 350–357. Knowledge Systems Institute, (June 1995).