COMPONENT-BASED SUPPORT FOR KNOWLEDGE-BASED SYSTEMS

Sabine Moisan
INRIA, BP 93, 06902 Sophia Antipolis, France

Keywords: Software engineering, components, frameworks, knowledge-based systems, inference engines.

Abstract: Software development of knowledge-based systems is difficult. To alleviate this task we propose to apply software engineering techniques. This paper investigates BLOCKS, a component framework for designing and reengineering knowledge-based system inference engines. BLOCKS provides reusable building blocks common to various engines, independently on their task or application domain. It has been used to build several engines for various tasks (planning, classification, model calibration) in different domains. The approach appears well fitted to knowledge-based system generators; it allows a significant gain in time, as well as it improves software legibility and safeness.

1 MOTIVATION

Our objective is to facilitate operational knowledge-based system (KBS) implementation and evolution. Knowledge-based systems are mainly composed of three parts: a knowledge base storing expertise in a particular domain, a fact base containing facts about an end-user problem in this domain, and an engine, written by a software designer, performing inferences to solve the end-user problem based on the expert knowledge. For instance, if the task is to classify objects within a taxonomy, the knowledge base contains a description of the class hierarchy and features, the fact base a set of observed features concerning yet unclassified objects, and the engine implements an algorithm that infers the classes of the objects by traversing the class hierarchy.

Building a KBS means to develop all the tools for both experts and end-users to interact with the system, such as inference engine, knowledge representation schemes, knowledge editors/verificators, etc. Each element by itself represents a great amount of code. Moreover, all elements must work together, although every one may evolve independently. So, not only developing new KBSs but also modifying them is a software engineering challenge. For both activities, designers have to convert a *cognitive model*, as expressed by experts, into a *software model* and eventually an *operational system*. This implies to bridge a large conceptual gap and recent software engineering techniques seem good candidates to support this task. Indeed, KBS software designers have little adapted support, much effort has been devoted to facilitate experts and end-users' tasks. Since the major modifications usually concern engines (changes in reasoning strategy) we focus this paper on a solution to allow easier building and reconfiguration of KBS engines by means of an extensible and reusable component framework.

For a long time now, frameworks have been considered as a powerful tool for designing and building complex software systems in a rather economical and flexible way. Following this line, our framework, named BLOCKS, provides designers with building blocks to create engines at a level of abstraction higher than ordinary programming languages. It helps cope with model changes, ensuring programming security and rapid code production. Moreover, it provides a way to compare different reasoning strategies.

After this introduction, section 2 describes our meta approach to the KBS life cycle. Section 3 details different engines developed for three different tasks. Finally section 4 compares our approach with some others, before concluding.

2 BEYOND KBS GENERATORS

Each knowledge-based system realizes a *problem-solving task* such as planning, classification, diagnosis, etc. Depending on target problems, on expert purposes or on implementation requirements, different versions of the same task may be necessary. Over

Moisan S. (2008).
COMPONENT-BASED SUPPORT FOR KNOWLEDGE-BASED SYSTEMS.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - AIDSS*, pages 295-300
DOI: 10.5220/0001702702950300

the time, domain evolutions also necessitate a series of versions for the same task.

For a given range of problem solving tasks, there exist commonalities, such as specific knowledge organization or contents. Sharing those commonalities, to a certain extend, has been the purpose of KBS *generators* or *shells*. Generators provide a panel of common elements to design a KBS, namely inference engine, knowledge representation schemes, verification tools, and various editors. Except for general rule-based shells (such as Jess), most generators are more or less dedicated to a given range of applications or to a given task. Such specialized generators are closer to expert ways of reasoning and often lead to more efficient KBSs. Generators properly meet expert modification needs at the cognitive level, since they support modification and maintenance of knowledge base contents and, to some respect, minor modification in reasoning strategies. At the software level however, when new functionalities are required in one of the elements provided by a generator, modifications are difficult, since important features are often hidden inside different pieces of code.

To improve code flexibility, we propose a "meta generator" approach to go one step further and to enable the *reuse and extension* of each element provided by a generator, i.e. inference engine, interface, knowledge base description language, knowledge verification tool, etc. This approach is implemented in a software platform, named LAMA which gathers several generic extensible toolkits to design, test, and modify these software elements. The platform provides a unified environment to design or to modify generators and to tune variants of them in order to fulfill specific requirements. The focus of this paper is the component framework for engine design, which is detailed below, but the platform also offers a framework for customising expert level languages, a framework for graphic user and expert interfaces and a library for knowledge verification. We mainly rely on frameworks to provide high level tested software architectures and implementations for each KBS element.

2.1 The BLOCKS Framework

BLOCKS is the core of the platform. It is an object-oriented framework, in the sense of (Johnson, 1997), written in C++, and rooted in our extensive experience of the design of various KBS generators for computer aided design, classification, or planning, in domains as different as civil engineering, astronomy, medicine, or biology. Briefly, components in BLOCKS correspond to interrelated classes, and more precisely to roots of class hierarchies. The structure of these (abstract) classes form patterns for describing the concepts involved in a task; generic functions or (abstract) methods of these classes constitute a kernel of basic instructions that can be redefined to implement a reasoning strategy. Designers can thus reuse or extend both the set of concepts or the algorithmic capabilities. These two aspects are of course strongly connected and have to be modified accordingly.

BLOCKS is composed of a common general layer and several task specific layers on top of it. The general layer consists of about 75 classes that implement generic features useful for a large range of KBSs: e.g., inference rules, structured frames, or history management. By specializing classes in the general layer, a designer may define task-customized layers to implement adapted task models. These layers contain only concrete classes, the instances of which will populate the knowledge bases, and methods or functions that will constitute reasoning steps in the algorithm of an engine strategy.

The proper generality level of the components was an important issue during our domain analysis. BLOCKS is not reduced to unsubstantial classes and generic functions, but it offers structured classes with rich behavior and relationships that reflect the usual interactions between concepts in KBS engines. Class interfaces are complete enough to cover most designer needs without modifications, but points of flexibility (*hooks*) have been foreseen, in particular in methods. Specialization, composition and hooks allow designers to fine tune engine behavior.

2.2 Use of BLOCKS for KBS Engines

As any class framework, BLOCKS not only supplies concrete and abstract classes that can be derived or composed, but also relationships among classes that can be extended, generic functions that can be parametrized by criteria (written as functions), and (abstract) methods that can be redefined. It thus provides a global organization of the classes, relationships, and functions that must be respected by designers. Enforcing this point is an important issue, which is outside the scope of this paper; it is addressed by model-checking techniques (Moisan et al., 2004).

Figure 1 illustrates the way BLOCKS is to be used. A *designer* implements a particular engine for a given task (1) by picking the needed classes, methods and functions from the general layer (or from an existing specific layer). The designer can reuse them "as is", compose or specialize them according to domain requirements. These operational classes, methods or functions are used to compose a new engine program (2) that corresponds to the chosen strategy. Roughly

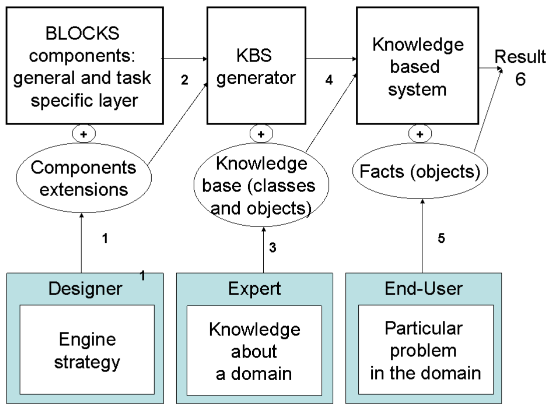

Figure 1: Generation of a KBS using BLOCKS.

speaking, each reasoning step in the engine corresponds to a method or function call.

An *expert* can then feed this engine with knowledge about his/her particular domain (3), guided by the other tools provided by the designer (language, editors...). Experts do not modify the engine procedure. Through the expertise-oriented language, they transparently introduce sub classes (domain concepts) and/or instances of the designer provided classes. This constitutes a knowledge base, which complements the engine, providing an executable knowledge-based system (4).

Then, *end-users* solve particular problems involving the given reasoning and expertise. End-users do not create classes, nor can they modify the engine strategy. They provide facts describing their problems (5) and run the system over these problems. During the execution, instances of the operational classes are created, modified or deleted by the system. The final result (6) ensues from the expertise and engine behavior. For instance, if the task is classification, the end-user problem is an object to classify and the result a list of candidate classes with attached likelihoods.

3 ENGINE REALIZATIONS

Over the past decades BLOCKS was used to design or modify different inference engines, for different generators. First, we re-wrote existing systems, that were initially developed from scratch; then, based on promising results, we systematically used the framework to develop new engines. We started by variants of planning engines. This intensive code (re)writing activity was the motivation of the platform approach. Thus, the development of the framework and the design of these engines were conducted in parallel. We then have addressed a completely different task, classification, resulting in two engines. In parallel, we developed a model calibration engine as an extension of our work in planning. These three tasks led to specific layers in BLOCKS.

Planning Engines. We have developed several KBS engines for a task called *program supervision* which consists in automating the use and correct combination of existing programs. This task mainly relies on planning techniques. It introduces the notions of *supervision operators*—corresponding in our case to programs or compositions of programs— that manipulate *data* (arguments of programs). Operators are combined in a plan to achieve a processing goal. We have developed a layer for this task and three variants of engines, based on a initial engine, OCAPI (Clément and Thonnat, 1993), initialy written in Lisp.

The first variant, PEGASE, performs pure hierarchical planning, as OCAPI, though introducing new operator and rule types. Thanks to this experience, we were able to define an initial version of a planning layer in BLOCKS. PEGASE has been successfully applied to domains such as galaxy identification in astronomical imaging (Thonnat et al., 1995), or vehicle detection in satellite imaging (Shekhar et al., 1997).

The second engine, MEDIA (Crubézy et al., 1997), integrates dynamic planning steps. It extends the planning layer of BLOCKS with new components, such as a *weak condition* concept, and it also defines more sophisticated rules. This experiment allowed us to improve the platform and in particular to tune the granularity of BLOCKS components for better reuse. It is a typical example of successful reuse, the gain in time had been dramatically demonstrated: developing MEDIA took a PhD student two months (for the coding part, after analysis) and reused more than 90 % of the components that were developed for PEGASE.

Finally, the PULSAR engine was an attempt towards combining hierarchical and dynamic operator-based planning methods. It provides a mechanism to handle *unordered compositions*, a new type of operator composition. For hierarchical planning aspects, it simply reused parts of the PEGASE engine. PULSAR has been applied to road obstacle detection and to medical imaging (van den Elst, 1996). It took a PhD student four months to implement the PULSAR engine and to test it on some examples. This is reasonable considering the fact that it was our first planner containing a combination of hierarchical skeletal and component based partial ordered planning, and thus it implied recoding and debugging some functionalities and data structures in the planning layer.

These engines share comon data structures and functions from the program supervision and the general layers. For instance, classes *Supervision Operator* or *Data* from program supervision layer are common to all three engines and generic function *select* from the general layer is used in all engine reasoning algorithm to select which planning operator to fire.

Classification Engines. Our first classification engine, named TACLE, was a C++ implementation of an previous Lisp engine. TACLE classifies an object in a predefined taxonomy (hierarchical description of all the possible classes of objects for a given domain). This attempt to tackle a completely new task was our first experiment in adding a new layer to BLOCKS. This layer primarily introduces an implementation of the theory of possibilities to take into account uncertainty on object attribute values and on rules. Figure 2 shows a few classes using this theory that have been derived from the BLOCKS general layer.

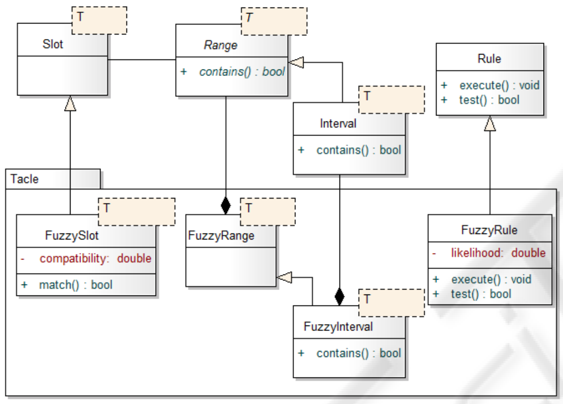

Figure 2: Some specializations for classification task.

It took us three months to implement it. TACLE used (directly or by specialization, see Figure 2) more than 85% of the general layer. It introduces 20 classes (half of them being very simple), among which 14 derive from BLOCKS components. This experience permitted us to fine tune the boundary between the general layer and specialized ones.

More recently, in collaboration with INRA, an engineer designed a new engine (OCEAN) to perform semantic image interpretation, i.e. to assign a meaning to data automatically extracted from images. This task implies to combine both a program supervision sub-task to plan image processing programs that extract interesting features, and a classification sub-task to search matching features in a taxonomy. OCEAN is still under construction. Up to now, we have introduced 15 (new and derived) classes, plus 8 directly reused from TACLE.

Model Calibration Engine. Model calibration is an essential step in modeling physical processes by means of mathematical equations. It consists in adapting numerical parameters of equations with respect to real measures, so that the simulation results of the numerical model fits the ground truth. This task shares a lot of concepts and reasoning steps with program supervision. So it has been easy to move to a new layer and a new engine (HYDRA) by specializing general layer items and reusing planning ones, with some tuning (see Fig. 3). In total, this engine introduces 13 classes and 9 major method definitions, one for each new/modified reasoning step, plus a few utilities. This engine was partly developed at Cemagref Lyon and tested in hydraulic model calibration, on quite different cases of French rivers (Vidal et al., 2005).

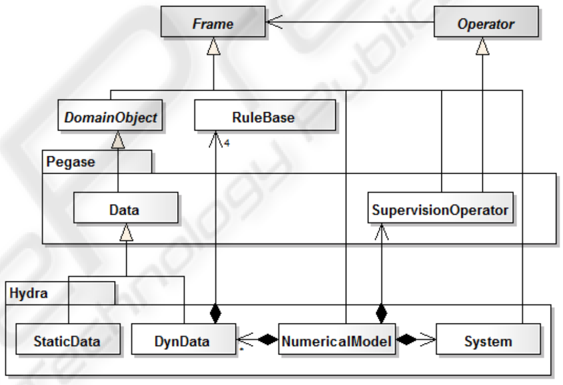

Figure 3: Major HYDRA class extensions.

Empirical Measures. The following table summerizes some figures about three engines, one in each task, just to give a rough idea about the coding effort which is necessary to build a new engine. BLOCKS itself is about 75 classes and 5,000 lines of C++ code (not including comments and support classes, as lists, strings,...). This table raises some remarks. First, PEGASE has been the most used of our engines, and it is the only one with a sophisticated history backtracking mechanism, hence its higher number of code lines. It is likely that OCEAN once completed will also reach similar complexity. On the other hand, HYDRA, which reuses a lot from PEGASE, necessitates less extra code. Second, the number of classes that are to be developed (brand new ones or derived from existing ones) is rather similar among engines. Third, an engine reasoning part, that is the algorithmics of its problem solving method, is on the order of a few hundreds of lines, which is a tractable size when it comes to modifications.

Engine	PEGASE	TACLE	HYDRA
Derived classes	16	15	11
New classes	2	5	2
Lines of (real C++) code	3,000	2,000	1,500
Reasoning part	850	150	250

4 RELATED WORK

The need to facilitate (re)writing of systems and to provide environments supporting that purpose has led to many works in Artificial Intelligence. These works mainly rely on reusability and customization, but they differ by the nature of what is reused or customized and by the techniques they use.

Most of them are interested in reusing knowledge itself (especially ontologies) or parts of reasoning strategies (termed problem solving methods). In the knowledge acquisition community, reusability often targets ontology management and task modeling, as in (Oussalah, 2003), PROTÉGÉ (Gennari et al., 2003), or Par-KAP (Nunes de Barros et al., 1997). Their objective is to help design knowledge *modeling* or knowledge *acquisition* tools, while we target *reasoning* KBSs. Following KADS (Schreiber et al., 1999) most systems focus on *formal* abstract models of ontologies and methods, whereas we propose reusable *operational* components. Technicaly, customization can be made possible through abstract component reuse, open-source approaches, as in Jess shell, or plug-ins, as in PROTÉGÉ. We rather rely on class *components* to be composed or derived.

Tools have been proposed, that intend to cover all steps of a KBS design (from cognitive model to implementation or simulation). We can cite DSTM (Trichet and Tchounikine, 1999), UPML (Fensel et al., 2003) or TASK (Talon and Pierret-Golbreich, 1996) that are dedicated to KBS design. Although such tools provide means to specify a KBS and even if they propose code implementation—by automatic translation of formal models— they do not offer real software development tools to designers of KBSs. In this sense they stand upstream from BLOCKS which promotes a software level composition of engine strategies from reusable operational components. We have started to study interoperability with some of these tools, relying for instance on standardized knowledge formats.

Some frameworks and libraries relying on component engineering more concerned with implementation issues exist, but they do not target knowledge-based systems; for instance, frameworks have been developed for agent platforms (Briot et al., 2002) or for learning tools, e.g., MLC++ (Kohavi et al., 1994) or WEKA (Witten et al., 1999).

Other systems address parts of the BLOCKS general layer or specific layers. In the first category we can cite the AROM platform (Page et al., 2001) for knowledge base edition which is close to the knowledge representation part of BLOCKS. In the second category, we find frameworks such as ASPEN (Chien et al., 2000) dedicated to planning/scheduling systems for space mission operations. Such frameworks rather correspond to a specific layer of BLOCKS without the notion of a general layer comon to several tasks.

This general layer is however important, it can be viewed as a meta-model of KBSs, based on well-established concepts to serve as the basis of KBS development. It can be compared to the UML Profile proposed in (Abdullah et al., 2007) or to the JavaDON architecture (Tomic et al., 2006). Both stand at he same meta level of KBS representation as BLOCKS also with a clear software engineering perpective. A main difference is that both rely uniquely on rules as inference mechanism, when we propose to mix a rule engine with other ones (e.g., planning or classification ones). There are also a few differences in goals (stress on Web applications for JavaDON), methods (UML profiling or open-source), or achievements (no code generation from UML Profile yet).

5 CONCLUSIONS

We propose to use software engineering techniques to improve knowledge-based system design. In particular, we have investigated a reusable and adaptable component framework to design KBS engines. The BLOCKS framework provides the necessary components for high-level design and implementation of (variants of) engines, based on reuse, composition and refinement. Yet, genericity is not at the expense of efficiency: for instance in a video understanding application, a planning engine generated with BLOCKS provides high versatility but accounts for less that 4% of the overall execution time. Our objective was operational code reuse, not only (formal) model reuse. This approach has proved well fitted to knowledge-based system engines and led to a significant gain in development time and in code readability. It has substantially simplified the creation of new engines, compared to previous implementations from scratch.

We are currently adapting BLOCKS components to support distributed knowledge-based systems and real-time performances, as needed by more and more applications. We use agent-based architecture and concurrent programming for these purposes. On a methodological side, we intend to investigate model driven engineering, to promote a seamless process from specification to realization of complex systems, by means of model transformations.

REFERENCES

Abdullah, M. S., Paige, R., Kimble, C., and Benest, I. (2007). A UML Profile for Knowledge-Based Systems Modelling. In *SERA'07: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 871–878, Washington, DC, USA. IEEE Computer Society.

Briot, J.-P., Charpentier, S., Marin, O., and Sens, P. (2002). A Fault-Tolerant Multi-Agent Framework. In *International Joint Conference on Autonomous Agents and Multiagent Systems*, Bologna Italy. ACM.

Chien, S., Rabideau, G., Knight, R., Sherwood, R., Engelhardt, B., Mutz, D., Estlin, T., Smith, B., Fisher, F., Barrett, T., Stebbins, G., and Tran, D. (2000). ASPEN - Automated planning and scheduling for space mission operations. In *6th International Symposium on Space missions Operations and Ground Data Systems (SpaceOps 2000)*.

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

Crubézy, M., Aubry, F., Moisan, S., Chameroy, V., Thonnat, M., and di Paola, R. (1997). Managing Complex Processing of Medical Image Sequences by Program Supervision Techniques. In *SPIE International Symposium on Medical Imaging*, Newport Beach, California USA.

Fensel, D., Motta, E., van Harmelen, F., Benjamins, R., Crubezy, M., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., Musen, M., Plaza, E., Schreiber, G., Studer, R., and Wielinga, B. (2003). The Unified Problem-Solving Method Development Language UPML. *Knowledge and Information Systems*, 5(1):83–131.

Gennari, J., Musen, M., Fergerson, R., Grosso, W., Crubezy, M., Eriksson, H., Noy, N., and Tu, S. (2003). The Evolution of Protégé: An Environment for Knowledge-Based Systems Development. *Int. Journal of Human-Computer Studies*, 58:89–123.

Johnson, R. E. (1997). Frameworks = (Components + Patterns). *CACM*, 10(40):39–42.

Kohavi, R., John, G., Long, R., Manley, D., and Pfleger, K. (1994). $\mathcal{MLC}++$: A Machine Learning Library in C++. In *Tools with Artificial Intelligence*.

Moisan, S., Ressouche, A., and Rigault, J.-P. (2004). Towards Formalizing Behavorial Substitutability in Component Frameworks. In *2nd International Conference on Software Engineering and Formal Methods*, pages 122,131, Beijing, China. IEEE Computer Society Press.

Nunes de Barros, L., Hendler, J., and Benjamins, V. (1997). Par-KAP: a Knowledge Acquisition Tool for Building Practical Planning Systems. In M.E. Pollack, editor, *15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 1246–1251.

Oussalah, M. (2003). Reuse in KBS: a components approach. *Expert Systems with Applications*, 24,(2):173–181.

Page, M., Gensel, J., Capponi, C., Bruley, C., Genoud, P., Zibelin, D., Bardou, D., and Dupierris, V. (2001). A New Approach in Object-Based Knowledge Representation: the AROM System. In *EA/AIE'2001*, Budapest, Hungary.

Schreiber, G., Wielinga, B., and Breuker, J. (1999). *KADS: A Principled Approach to Knowledge-Based System Development*. Academic Press, London.

Shekhar, C., Burlina, P., and Moisan, S. (1997). Design of Self-Tuning IU Systems. In *DARPA Image Understanding Workshop*, volume 1, pages 529–536, New Orleans, LA.

Talon, X. and Pierret-Golbreich, C. (1996). TASK: from the specification to the implementation. In *8th International Conference on Tools with Artificial Intelligence - ICTAI*.

Thonnat, M., Clément, V., and Ossola, J. C. (1995). Automatic Galaxy Description. *Astrophysical Letters and Communication*, 31(1-6):65–72.

Tomic, B., Jovanovic, J., and Devedzic, V. (2006). Javadon: an open-source expert system shell. *Expert Syst. Appl.*, 31(3):595–606.

Trichet, F. and Tchounikine, P. (1999). 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 (ESWA)*, 16(2):105–120.

van den Elst, J. (1996). *Modélisation de Connaissances pour le Pilotage de Programmes de Traitement d'Images*. PhD thesis, Université de Nice.

Vidal, J.-P., Moisan, S., Faure, J.-B., and Dartus, D. (2005). Towards a Reasoned 1-D River Model Calibration. *Journal of Hydroinformatics*, 7(2):79–90.

Witten, I., Frank, E., Trigg, L., Hall, M., Holmes, G., and Cunningham, S. (1999). Weka: Practical machine learning tools and techniques with Java implementations. In *ICONIP/ANZIIS/ANNES'99 Int. Workshop: Emerging Knowledge Engineering and Connectionist-Based Info. Systems*, pages 192–196.