HAL
archives-ouvertes.fr

BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems

Sabine Moisan, Annie Ressouche, Jean-Paul Rigault

▶ **To cite this version:**

Sabine Moisan, Annie Ressouche, Jean-Paul Rigault. BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems. Informatica, Slovene Society Informatika, Ljubljana, 2001, Special Issue on Component Based Software Development, 25 (4), pp.7. hal-01861673

HAL Id: hal-01861673
https://hal.inria.fr/hal-01861673

Submitted on 25 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems

Sabine Moisan[1], Annie Ressouche[1], Jean-Paul Rigault[1,2]
[1] INRIA Sophia Antipolis, France
Phone: +33 4 92 38 78 47, Fax: +33 4 92 38 79 39
E-mail: {Sabine.Moisan, Annie.Ressouche}@sophia.inria.fr
[2] I3S Laboratory, University of Nice Sophia Antipolis, France
Phone: +33 4 92 96 51 33, Fax: +33 4 92 96 51 55
E-mail: jpr@essi.fr

Keywords: component framework, behavioral model, model checking, artificial intelligence

Edited by: M. B. Juric

Received: **Revised:** **Accepted:**

BLOCKS *is an answer to the software engineering needs of the design of knowledge-based system engines. It is a framework composed of reusable and adaptable software components. However, its safe and correct use is complex and we supply formal models and associated tools to assist using it. These models and tools are based on behavioral description of components and on model checking techniques. They ensure a safe reuse of the components, especially when extending them through inheritance, owing to the notion of behavioral refinement.*

1 Introduction

In the design of Knowledge-Based Systems (KBS) more attention has been paid to cognitive issues than to software engineering ones. Yet, *software* quality (reusability, maintenance, evolution, and safety) is also an important issue for such systems. That is why we have developed a generic multi-level approach to KBS development relying on best software engineering practices. A major outcome is a component framework enriched with models and tools enforcing the correct use of the framework.

A Knowledge-Based System basically consists of an inference engine, a knowledge repository (aka Knowledge Base), and a fact base. Each of these three parts is the realm of one particular type (or role) of actor. In this paper we focus on the role of the *designer*, the one who develops KBS engines.

The notion of *KBS generators* (or shells) emerged in the late 80's [17]. A KBS generator addresses a given activity (e.g., diagnosis, classification) but it is domain-independent: its KBS instances apply to various domains (e.g., classification of cardiologic diseases, of astronomic objects, of biological organisms). KBS generators take advantage of the cross-domains similarities by abstracting the common artificial intelligence concepts and by gathering representation techniques within a unique environment.

Whereas generators aim to meet experts or end-users needs (e.g., they help them manage knowledge base evolution and maintenance), they provide little help for the designer as far as Software Engineering is concerned. Therefore we promote generic tools for producing KBS generators. Adding such a level improves versatility but increases complexity. This paper proposes methods and tools to assist the designer in implementing Artificial Intelligence (AI) techniques in an efficient, versatile, reusable, and maintenable way.

To face the corresponding software engineering challenge (essentially a reusability problem), a collection of software engineering best practices have been prescribed: object-oriented modeling (UML) and programming (C++ and Java), component-oriented framework [3, 10], behavioral modeling with associated proofs and simulations. In a KBS, the primary element that is likely to evolve is the inference engine. That is why this paper focuses on the design, simulation, and validation of engines.

In the sequel we first describe our engine design

framework, named BLOCKS[1] (section 2). Then we present the static model and the notion of a component in BLOCKS (section 3). Section 4 is devoted to the component behavioral model and the associated verification techniques. We finally discuss the scope and the benefits of our approach (section 5).

2 General Description of BLOCKS

This paper concentrates on BLOCKS which is part of a wider software platform providing designers with a set of generic toolkits. In addition to BLOCKS (components for engine design) the platform offers compiler generators for knowledge description languages, and several libraries (for graphic user interfaces, for knowledge base simulation and verification). The task of the designer is to select, adapt, and assemble components from these toolkits into a customized KBS generator, which can then be used to develop KBS applications.

The objective of BLOCKS is to help designers create new engines and reuse or modify existing ones without extensive code rewriting. Thus the components of BLOCKS stand at a higher level of abstraction than programming language usual constructs.

The framework consists of around 60 (C++) classes. About a dozen of them implement basic data structures (lists, sets, maps...). The remaining classes are dedicated to knowledge representation artefacts such as the classical AI notions of *frame* and of *rule* [8]. As a matter of example, class `Rule` is composed of a set of conditions and a set of actions that are to be executed when the conditions are true (see figure 1).

The methods of BLOCKS classes are used by the designer to construct new KBS engines. To continue with the same example, class `Rule` sports two fundamental methods: one to test the conditions, the other to execute the actions. Calls to these methods will appear in the code of rule engines. For instance, a classical forward-chaining engine loops over three phases: finding applicable rules (call `Rule::test_conditions`); selecting a rule for execution (conflict resolution specific strategy, written by the designer); execution of the chosen rule (call `Rule::execute_actions`).

The framework is rooted in our extensive experience with designing various KBS engines, for activities as diverse as computer aided design, classification, or planning and in domains as different as civil engineering, astronomy, medicine, finance, etc. This has been the basis for a *domain analysis* that allowed the major concepts of BLOCKS to emerge. A crucial design decision was to determine the proper generality level of the framework components. Too much generality is not suitable for efficiency, whereas too specific components, though easily applicable, are hardly reusable. Our solution was to restrict the range of targeted activities: we choose planning and classification, merely because they are useful in our current applications.

The analysis has been an iterative process with three main steps:

- abstract modeling of existing engines using formalisms such as UML [18]; this led to the definition of the knowledge representation classes;
- completing classes and detailing their behavior; this has been a major step for identifying common concepts and methods behavior, their roles in problem solving, and their organization;
- modeling control to define sequencing of method calls in engines.

BLOCKS is divided into several layers: the *support* layer contains generic and abstract features (abstract classes and methods, and generic functions) useful for any kind of engine. By specializing the classes in the support layer, the designer may define new layers dedicated to specific activities. These layers contain concrete classes, the instances of which will populate the knowledge bases.

3 A Component View of BLOCKS

In BLOCKS we define a component as the realization of a sub-tree of the class hierarchy: this complies to one of Szyperski's definitions for components [20]. At the framework top level, there are presently three such components that the designer may compose or extend. For this to be possible, the designer needs information about component properties. For it to be safe, he or she should commit to some protocol. For forcing it to be safe, we offer automatic proof and validity checking tools.

3.1 Components in BLOCKS

The three high level components are associated with the initial sub-trees of classes `Frame`, `Rule`,

[1] Basic Library Of Components for Knowledge-based Systems

and State, corresponding to major KBS concepts. Frames describe pieces of knowledge as static structures, composed of attributes which in turn are composed of sub-attributes or "facets" (declarative or procedural). Rules describe pieces of knowledge as dynamic inferences in the form of conditions/actions patterns. States store the history of the problem solving process.

The designer both adapt the components and writes the glue code of engines. To achieve a given strategy he/she will (non-exclusively) use these components directly, or extend the classes they contain by inheritance, or compose the classes together, or instantiate new classes from predefined generic[2] ones. Among all these possibilities, class derivation is certainly the most frequent one. It is also the one that raises the trickiest problems. In the sequel we shall mainly concentrate on it.

Let us continue with our example: the Rule class in BLOCKS (figure 1) is composed of conditions and actions which originally do not take into account fuzzy values. Thus, as mentioned in section 2, it can be used by a simple rule engine. To cope with activities requiring fuzziness, the designer must introduce a FuzzyRule class as a derivative of Rule. Relying on the static information of the class diagram of Rule (signatures of methods and associations among classes), the designer obtains the inheritance graph shown on figure 1. But this static information is not sufficient to ensure a safe use of the framework. Indeed, in the example, the designer must also redefine–in a "semantically acceptable" way–methods test_conditions and execute_actions.

3.2 Protocol to Use the Framework

As previously mentioned, safe use of the framework requires that a *protocol* be specified. This protocol of use is defined by two sets of constraints. First, a static set enforces the internal consistency of class structures; for instance, in C++, class derivation and composition demand a scaffolding of structure-dependent construction/destruction operations. The static nature makes it easy to generate the necessary information at compile-time.

A second set of constraints describes dynamic method requirements:

1. legal sequences of method calls; for instance, Rule requires that test_conditions be invoked before calling execute_actions;
2. constraints on the operations that a component expects from other components; in the example, the execute_actions method expects actions to sport an execute method; this is hardly obvious on the class diagram(s);
3. specification of internal behavior of methods;
4. specification of the valid ways to redefine method behavior in derived classes.

[2] "template classes" in C++

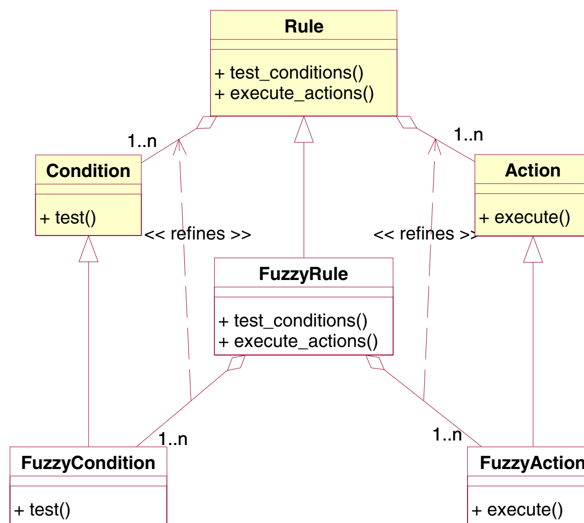

Figure 1: Rule and FuzzyRule classes: above the original classes, below the derived ones.

These dynamic aspects are more complicated to express than static ones, they are error-prone, and there is no tool (as natural as a compiler for the static case) to handle and check them. While items 1 and 2 can be partially addressed by classical UML models (class diagrams and Statecharts), the last two items are more challenging. We shall propose a solution in section 4.

3.3 Realizing the Component Protocol

To implement the protocol of the previous section BLOCKS applies three non-exclusive techniques.

First, well-known design patterns [9] make it possible to create polymorphic objects (abstract factory, virtual constructor, singleton, prototype), to traverse complex data structures (iterator, visitor), and to implement polymorphic algorithms (strategy). This helps clarify the software architecture, but it seldom is a complete solution.

Second, we use meta-programming [12], namely the OpenC++ meta-object protocol [4, 5]. This helps generate the language-dependent "scaffolding" of constructors requested for frame derivation. It also allows to implement some specific "aspects" [13] of frames such as introspection or persistence. However meta-programmation is complex. Moreover the knowledge about components is external to the components, a risk of inconsistent evolution.

Therefore, third, the knowledge for using, deriving, and composing is embedded into the components themselves. This allows static as well as dynamic verifications relying on this knowledge.

The first two techniques are out of the scope of this paper. We focus on representing and embedding information about behavior of components and methods. There is no complete and consensual technique for this: for instance, in JavaBeans, the embedded knowledge is rather poor; in CORBA, the IDL is external to the components and is not much richer. The next section presents our solution.

4 Behavior Description and Behavior Refinement

In order to reuse BLOCKS components in a safe way, we define a mathematical model providing consistent description of *behavioral entities*. Behavioral entities are whole components, sub-components, or single methods. Such a model complements the UML approach and allows to specify the class and method behavior with respect to class derivation. We also propose a *hierarchical* specification language to describe the dynamic aspect of components both at the class and method levels. Finally we define a *semantic* mapping to bridge the gap between the specification language and its meaning in the mathematical model.

In this paper we just intend to give the flavor of the formal models.

4.1 Mathematical Model of Behavior

We have chosen input/output labeled transition systems [15] as a basis for our mathematical model. Since these systems are a special kind of finite state machines (automata), we shall denote them LFSM for short in the rest of the paper. In our model a LFSM is associated with a behavioral entity; each transition has a label representing an elementary step of the entity, consisting of a *trigger* event (input condition) and the *action* to be executed when the transition is fired.

LFSMs are particularly well suited to check temporal logic properties. Temporal logic easily expresses assertions about behavior. Formulae of this logic concern either the states of the model or its executions[3]. Moreover, tools and proof environments are available to perform temporal logic checking on LFSM [11]. The major drawback of model checking is a possible explosion of the state space. Although some tools use symbolic model checking methods to cope with it, an obvious method to push back the bounds of possibility is to use the natural decomposition of the system. Hence, our specification language provides a hierarchical description of behaviors that allows to merge symbolic and compositional approaches.

We substitute LFSM for regular UML Statecharts to represent the state behavior of a class as well as of a method.

In the object-oriented approach, the static semantics of specialization (aka class derivation, or subtyping, or extension) usually obeys the classical Substituability Principle [14]. To enforce behaviorwise *safe* derivation, the same principle should apply to the dynamic semantics of a behavioral entity–such as either a whole class, or one of its (redefined) methods.

If P and Q are LFSMs denoting respectively some behavior in a base class and its redefinition in a derivative, we seek for a relation $Q \preceq P$ stating that "Q extends P in a safe way". To comply with inheritance, this relation must be a preorder sufficient to capture the notion of "correct extention of behavior".

Q *simulates* P iff we can build a relation H that relates each state of P to a state of Q so that for two related states p an q, every successor of p is related to some successor of q with a transition bearing compatible[4] labels (trigger/action). The definition of simulation is local since the relation between two states is based only on their successor states. As a result, it can be checked in polynomial time. Intuitively, if Q simulates P then any valid input/output sequence (trace) of P is also a trace of Q. Thus Q can be substituted for P, for all purposes of P. Therefore, the extensions in Q do not jeopardize the behavior of P.

[3] Temporal logic is based on first order logic and has specific temporal operators to express properties holding for a given state, for the next state, eventually for a future state, or for all future states. We can also express that a property holds for all the executions starting in a given state or that it exists an execution satisfying a given condition.

[4] Two labels are compatible if they are equal once restricted to the intersection of the LFSM alphabets.

For $\preceq$ we choose the notion of "simulation preorder", i.e., $Q \preceq P$ iff there is a simulation relation H such that $H(q_0, p_0)$, where q_0 and p_0 are respectively the initial states of Q and P. Relation $\preceq$ is a preorder over LFSMs and it preserves satisfaction of the formulae of a subset of temporal logic, expressive enough for most verification tasks ($\forall CTL$ [11]). Moreover, this subset has a practicable model checking algorithm.

To capture the notion of safe extensibility for components, we define a relation ($\sqsubseteq$): if A and B are two classes, B $\sqsubseteq$ A iff B derives from A and the LFSM associated with B simulates the one associated with A. The relation is also defined for method behavior: $m \sqsubseteq m\prime$ iff the LFSM associated with m simulates the one of $m\prime$.

With such a model, the description of behavior matches the class hierarchy. Hence, class and method refinements are compatible and consistent with the static description: checking dynamic behavior may benefit from the hierarchical organization.

4.2 Behavior Description Language

In addition to the previous mathematical model, we propose a specification language. This language, very similar to the *Argos* graphical language [15], is also automata-based. It is easily compiled into finite state machines and it supports existing verification methods and tools. Programs written in this language operationally describe behavioral entities, we call them *behavioral programs*.

Behavioral programs use simple automata as a primitive construct. Labels correspond to input/output *events* which determine how the entity changes its state. The notion of event is abstract; in the language it is just represented by a name and, thus, it may receive various interpretations. For instance, it may be associated with the code of a method or with another behavioral program.

The language defines three main constructs. The first one is *parallel composition* (noted $P \parallel Q$). It is a symmetric operator which behaves as the direct product of its automata operands: transitions triggered by the same input are fired simultaneously and their outputs are unioned. Second, *local event declarations* allow to declare events local to a (behavioral) entity (when a local event is emitted, it can trigger transitions only in its own entity). Parallel composition combined with local event declarations makes it possible to represent communication between subprograms. Third, the *refinement* operator is similar to its Statecharts counterpart (definition of hierarchical states), except that it cannot break the hierarchical structure of programs and states. The states of an entity may be decomposed into behavioral sub-entities. This operator makes it possible to express interrupts, exceptions, and normal termination of (sub)programs.

This language offers a syntactic means to build programs that reflect the behavior of BLOCKS components. Nevertheless, the soundness of this approach implies a clear definition of the relationship between behavioral programs and their mathematical representation as LFSM (section 4.1). Let $\mathcal{P}$ denote the set of behavioral programs and $\mathcal{L}$ the set of LFSMs. We define a *semantic* function $\mathcal{S} : \mathcal{P} \longrightarrow \mathcal{L}$ that is stable with respect to the previously defined operators (local events, parallel, and refinement).

As a consequence, the language exhibits a fundamental *composition property*. This property is the key to simplify model checking. For instance if we have proved that $P_1 \sqsubseteq P_2$, then we can infer that $P_1 \parallel Q \sqsubseteq P_2 \parallel Q$, for any possible Q. Thus, compositionality provides a hierarchical means to verify properties.

4.3 Example: Adding Fuzziness to a Rule Engine

Let us apply the previous model to a simple rule engine, involving classes `Rule` and `FuzzyRule` (figure 1).

We can show that the behavioral program of `FuzzyRule` is a safe extension of the one of `Rule` (`FuzzyRule` $\sqsubseteq$ `Rule`). First, as can be seen on figure 2, the `FuzzyRule` behavior diagram is identical to the `Rule` diagram except that `FuzzyCondition` is substituted for `Condition` and `FuzzyAction` for `Action`. This diagram expresses the dynamic behavior of class `Rule` with respect to the correct sequence of method calls: `test_conditions` must be called before `execute_actions`.

Second, since `Rule` and `FuzzyRule` are composite classes, we must check the behavior of their parts. Thus we consider classes `Condition` and `FuzzyCondition`. Their behavorial programs are displayed in figure 3. In this simple example, it is easy to see that `FuzzyCondition` $\sqsubseteq$ `Condition`: `FuzzyCondition` derives from `Condition` and `FuzzyCondition` trivially simulates `Condition` (they are associated with identical LFSMs). The same holds for `Action` and `FuzzyAction`.

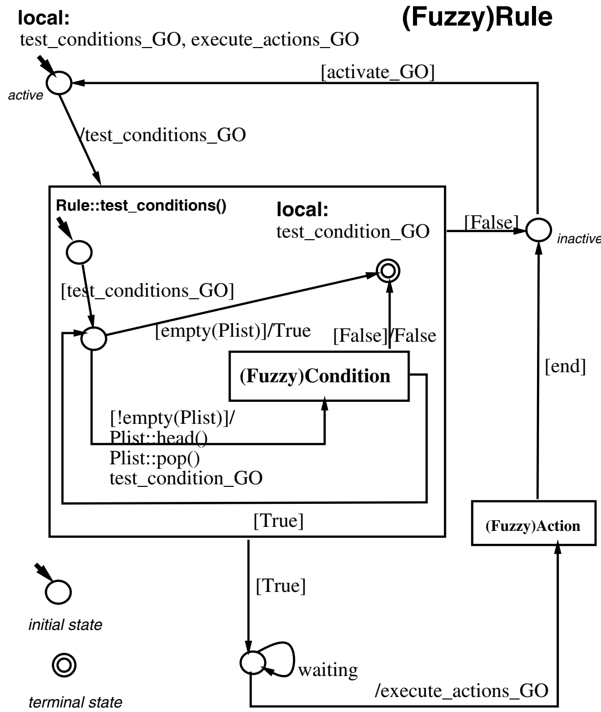

Figure 2: Rule and FuzzyRule behavior description. *Rectangular boxes represent refinement and the keyword* ***local*** *denotes local events. Note that we had to introduce events to trigger method calls (e.g.,* `test_condition_GO`*).*

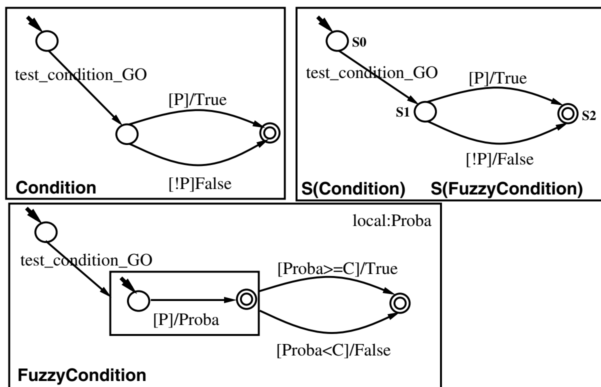

Figure 3: `Condition` and `FuzzyCondition` behavioral programs and semantics. *According to the semantics of refinement and encapsulation, it turns out that* `Condition` *and* `FuzzyCondition` *are associated with identical LFSMs. We recall that* $\mathcal{S}$ *is the semantic mapping of section 4.2.*

Hence, according to the composition property, we can deduce that `FuzzyRule` is substituable to `Rule` (`FuzzyRule` $\sqsubseteq$ `Rule`). Compositionality is indeed the way to avoid state explosion in this kind of models.

In this example, the proof is straightforward. In more complicated cases though, the proof may be less obvious, but tools are available to run it automatically.

Relying on this proof, the designer can safely implement the methods; he/she also has to modify the glue code of the engine, especially the conflict resolution strategy (section 2), e.g., to select the rule with the highest likelihood. The resulting rule engine will now accept fuzzy rules[5].

5 Discussion

5.1 Components and Frameworks

Both Software Engineering (SE) and Artificial Intelligence (AI) have an interest in component models. However they have different views on components and, hence, on reusability. SE tools focus on reusing code, analysis and design patterns, or software architectures. Few, if any, existing component frameworks go as far as ensuring correct use through a proof system. On the other hand, several AI approaches have been proposed to reuse *knowledge* components such as abstract problem-solving methods or ontologies [16, 19, 1]. They often manipulate formal descriptions but they usually remain at the knowledge level, thus they do not help producing code.

AI research has already proposed generic tools that cover all steps of KBS design (from cognitive model to implementation or simulation). We can cite DSTM [22] or TASK [21] that are dedicated to KBS design, although with different techniques and approaches. DSTM aims at prototyping a cognitive model before implementing it and, thus, it is more expert-oriented. TASK proposes different languages for the various steps of KBS design, and in particular a formal specification language. Such generic tools are very powerful since they are applicable across domains and activities, but their use may be difficult. Our work follows a similar line, with a stronger software engineering flavor.

5.2 Verification of KBS

In AI, the most common verification adresses the internal consistency of knowledge bases and, of course,

[5] Of course the other elements of the KBS generator (such as knowledge description language and expert interfaces) must be adapted accordingly: our platform provides the necessary toolkits. By assembling all these elements, the designer produces a new generator. Afterwards, experts can fill in different knowledge bases, in order to produce new KBS instances.

our platform provides tools for such verification. Usually, it is on the final KBS that verification is performed. It is too late since, at this time, all the KBS elements (domain knowledge, engine strategy, or even implementation artefacts) have been blended together. Hence, each verification process has to sort out its elements of interest. On the contrary, we promote high separation of concerns, i.e., we separate the engine design phase from the KBS one. The corresponding tools are also separated.

Some systems verify the KBS consistency against its domain and activity models. This verification generally relies on theorem proving techniques, using either an embedded theorem prover as in TASK or applying an external tool like KIV [7]. We have not yet investigated such verifications, but we expect that model checking could also be applied.

The Software Engineering issue of verifying that a KBS properly uses its generator features is often assumed and seldom performed. Our generic approach introduces such a verification. It corresponds to *usage verification* of a complete protocol of use (both static and dynamic properties). For this purpose, we use model checking instead of theorem proving, since it is adapted to our finite state machine model, it can be made automatic, and it can also automatically produce code for refined entities (furthermore this code will be correct, by construction).

5.3 Run-time Verification and Simulation

The designer can use our specification language to describe classes and methods behavior through a dedicated interface. The corresponding programs can serve both formal and practical aims.

On the formal side, the composition property makes it possible to apply model checking techniques in an incremental way. We have experimented with several tools. *EsterelStudio*[6] is a powerful environment to describe, simulate and verify reactive systems. However, its underlying paradigm (the synchrony hypothesis [2]) restricts the type of communication. By contrast, *Ptolemy*[7] is an open (meta-)tool for heterogeneous modeling and simulation. In particular, the user can introduce new models of communication. For this reason, we are going to customize Ptolemy; this will provide a simulation tool and a front-end for model checkers.

[6]from Esterel Technologies Company, http://www.esterel-technologies.com

[7]available at: http://ptolemy.eecs.berkeley.edu

On the practical side, as we already mentioned, our specification language can be used to generate (correct) code. The generated code can provide either skeletal implementations of methods, simulation code, and run-time trace facilities. Moreover, by embedding the code of behavioral programs in their components, we can achieve run-time verification.

6 Conclusion

We have experienced that framework technology can be adapted to the design of knowledge-based system engines. Such an approach allows a significant gain in development time. For instance, two years ago, we had to design a new planning engine [6]. Once the analysis completed, the implementation only took two months (instead of about two years for a similar former project started from scratch) and more than 90 % of the code was composed of existing components. Another experiment (for the classification activity) led to almost the same measurement.

However, the protocol to use the framework is complex and the static modeling (*à la* UML) is not sufficient to prevent the designer from fatal misuse. To this end, we assist the designer by modeling the behavior of components, thus permitting automatic verification during class derivation and composition. The model has also a pragmatic outcome: it allows the simulation of resulting KBS engines and the generation of code, of run-time traces, and of run-time assertions.

This behavioral formalism relies on a mathematical model, a specification language, and a semantic mapping from the language to the model. This lays the foundation for model checking and simulation tools. The model supports multiple levels of abstraction, from highly symbolic (just labels) to merely operational (pieces of code). Moreover this model is original in the sense that it covers both static and dynamic properties of components. To use our formalism, the designer has only to draw simple graphs with a (yet to be) provided graphic interface, oblivious of the underlying models and their complexity.

The same idea could be applied to other component frameworks, outside AI. Our approach gathers techniques from several Computer Science domains seldom intersecting each other: real-time and reactive systems, object-oriented paradigm, and knowledge-based systems. This work can be considered as a successful example of multidisciplinary integration.

References

[1] V.R. Benjamins, B. Wielinga, J. Wielmaker, and D. Fensel. Brokering Problem Solving Knowledge at the Internet. In *EKAW'99, European Knowledge Acquisition Workshop*, volume 1621 of *LNAI*. Springer-Verlag, 1999.

[2] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honor of Robin Milner*. MIT Press, 2000.

[3] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, and M. E. Fayad. Object-Oriented Frameworks: Problems & Experiences. In R. Johnson M. Fayad, D. Schmidt, editor, *Building Application Frameworks: Object Oriented Foundations of Framework Design*. John Wiley, 1999.

[4] J. Cavarroc, S. Moisan, and J-P. Rigault. Simplifying an Extensible Class Library Interface with OpenC++. In *OOPSLA'98, Worshop on Reflective Programming in C++ and Java*, 1998.

[5] S. Chiba. A Metaobject Protocol for C++. In *OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM Press, 1995.

[6] M. Crubézy. *Pilotage de programmes pour le traitement d'images médicales*. PhD thesis, Université de Nice Sophia Antipolis, 1999.

[7] D. Fensel, A. Schönegge, R. Groenboom, and B. Wielinga. Specification and Verification of Knowledge-Based Systems. In *Workshop on Validation, Verification and Refinement of Knowledge-Based Systems, ECAI*, 1996.

[8] R. Forsyth. *Expert Systems : Principles and Case Studies*. Chapman and Hall, 2nd edition, 1989.

[9] E. Gamma, R. Helm, R. Johson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[11] E. M. Clarke Jr., O.Grumberg, and D.Peled. *Model Checking*. MIT Press, 2000.

[12] G. Kiczales, J. de Rivière, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97*, volume 1241. Springer-Verlag, 1997.

[14] B. Liskov and J. L. Wing. A New Definition of the Subtype Relation. In *ECOOP'93*, volume 707 of *LNCS*, pages 119–141. Springer-Verlag, 1993.

[15] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Composition. *LNCS: Concur*, 630, 1992.

[16] M. A. Musen, S. W. Tu, H. Eriksson, J. H. Gennari, and A. R. Puerta. PROTEGE-II: An Environment for Reusable Problem-Solving Methods and Domain Ontologies. In *IJCAI*, Chambéry, August 1993.

[17] M. Richer. An evaluation of expert system development tools. *Expert Systems*, 3(3):166–182, July 1986.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

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

[20] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[21] X. Talon and C. Pierret-Golbreich. TASK: from the specification to the implementation. In *8th IEEE Int. Conf. on Tools with Artificial Intelligence*, pages 80–88. IEEE Computer Society Press, 1996.

[22] F. Trichet and P. Tchounikine. DSTM: a Framework to Operationalize and Refine a Problem-Solving Method Modeled in Terms of Tasks and Methods. *Int. J. of Expert Systems With Applications (ESWA)*, 16(2):105–120, February 1999.