On the use of knowledge representation techniques for modeling software architectures

Kim Mens*

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
E-mail: kimmens@vub.ac.be
Phone: +32 2 629 35 81

Michel Wermelinger†

Departamento de Informática
Universidade Nova de Lisboa
2825-114 Caparica, Portugal
E-mail: mw@di.fct.unl.pt

Abstract

We take the position that it could prove worth to reconcile ideas of the knowledge representation and software architecture research communities. Many existing knowledge representation techniques and formalisms seem to exhibit a lot of potential for representing different aspects of software architectures. To illustrate our case, we show how the theory of *conceptual graphs* could be a useful candidate to describe software architectures, to model architectural styles and patterns, and to serve as a formal foundation for compliance checking of architectures to architectural styles, as well as for checking conformance of the implementation of a software system to its architecture.

Topics addressed: architecture description languages, formal foundations, architectural styles.

Keywords: architectural style, compliance checking, conformance checking, conceptual graph, canonical graph, canonical formation rule, formula operator.

*Research funded by the Brussels' Capital Region (Belgium) and Getronics Belgium.

†Research funded by Laboratório de Modelos Computacionais da Fundação da Faculdade de Ciências da Universidade de Lisboa.

Reconciling software architecture with knowledge representation

Historically, one of the major goals of artificial intelligence (AI) research has been to investigate how knowledge, in its broadest sense, can best be modeled and represented. *Knowledge representation* and processing is essential to many subfields of AI, of which expert systems and natural language processing are prominent examples. Even in the domain of software engineering, knowledge representation has many applications, such as software comprehension and information systems, to name just a few. We claim that software architecture research also can benefit from many of the results achieved by the knowledge representation community. In fact, this does not come as a surprise, since knowledge representation is about building mental models of some problem domain. Software architecture can be considered a special case of this, as it is about building high-level intuitive abstractions of some software system. Many knowledge representation techniques and formalisms exist that could prove useful to represent different aspects of software architecture. Providing a complete overview is outside the scope of this paper. We only mention two promising techniques here: ontologies and conceptual graphs.

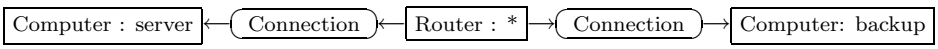
Deridder and Wouters [1] make a case for the application of **ontologies** in the domain of software engineering. They state that by integrating techniques and formalisms from the domain of computer linguistics and AI (in particular, ontologies and ontology-related techniques) in existing CASE tools, the software development process may significantly be enhanced. An ontology-based experiment was conducted to reverse engineer UML diagrams from an existing application. We believe a similar experiment could be set up to reverse engineer the architecture of an existing system.

As a second example, in the remainder of this paper, we explore the theory of **conceptual graphs** (CG) [5] as a knowledge representation formalism that seems suited to model and check software architectures.

Architectures as conceptual graphs

Conceptual graphs are finite and bipartite graphs, where the two kinds of nodes represent *concepts* and *relations* [5]. A concept is constituted by a concept type and the generic (or an individual) marker, separated by a colon. An n-ary relation is composed of a relation type and n arcs linking the relation to the concept arguments. Concept nodes are depicted with rectangles, relation nodes as round-corner rectangles.

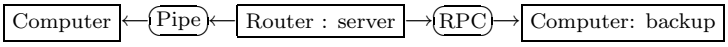
Representations of current day architecture description languages closely resemble such graphs, both visually and intentionally, since the explicit representation of the relationships between the system components is a basic tennant of software architecture. The nature of the components and the relations depends on the chosen architectural view [2]. For example, the following conceptual graph for the physical view states that the computer with hostname "server" is connected via an unidentified router to the computer called "backup":



where '*' is the generic marker and is henceforth omitted.

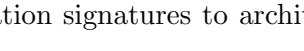
Architectural styles as canonical basis

The bare syntactical definition of conceptual graphs does not prevent the "wrong" use of types and markers. For example,



is a perfectly valid conceptual graph, although "pipe" and "RPC" are relationships of the logical view, and "server" is the identification of a computer. To prevent meaningless graphs, CG theory includes the notion of canon (further refined in [6]), which provides an ontological basis stating the restrictions on the possible concepts and relations. The canon consists of a concept type hierarchy, a relation type hierarchy, a set of markers, a conformity relation stating for each individual marker all the concept types it is compatible with[1], and an initial set of meaningful graphs, called the *canonical basis*.

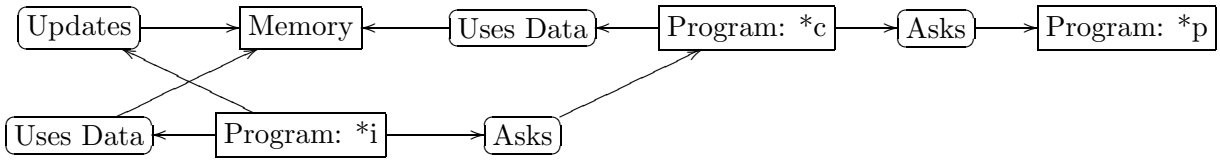
The set of all *canonical graphs* is obtained by applying *canonical formation rules* to the canonical basis. As an example, the concept specialisation rule allows to replace a type by a subtype or to replace the generic marker by an individual marker, if the new type/marker pair belongs to the conformity relation. Such rules guarantee that the resulting graph is meaningful.

The graphs in the canonical basis can be used to represent different kinds of patterns, ranging from simple relation signatures to architectural styles. 

is an example of the former, stating that the "pipe" connector can only link two components that are a subtype of "filter".

[1] By definition, the generic marker is compatible with all types.

As for styles, consider a simplified version of the interpreter style [4], where a program i is interpreting another one, using a memory to represent the state of the interpreted program p, and an auxilliary control program c to determine from p and the memory which is the next instruction to execute. The style is therefore a particular pattern of relationships between three programs and a memory, as shown next, where '*v' is the CG notation for named generic markers (i.e., variables).



Summing up, the canonical basis provides a catalog of patterns and styles, and style compliance can be formally defined as a derivation of the architecture from the canonical basis using the canonical formation rules.

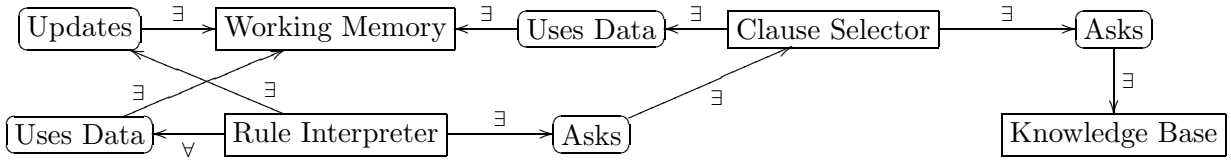
Conformance checking as formula operator

CG theory [5] defines a formula operator which translates a conceptual graph into a logic formula. This operator could be useful as the formal foundation for an algorithm to check conformance of the implementation of a software system to its architecture. First, we would use the formula operator to translate the conceptual graph representing the architecture into a logic formula. In this translation, conceptual relationships are replaced by high-level implementation relationships and the concepts are replaced by logic variables. Next, these variables should be instantiated with the concrete implementation artifacts which they represent. Evaluating the resulting logic formula corresponds to checking conformance of the implementation to its architecture.

As a further refinement, a concept (which corresponds to an architectural component) does not necessarily need to be instantiated with a *single* implementation artifact, but may correspond to a collection of such artifacts. This is easily formalized by the theory of conceptual graphs, where concepts may refer to sets of elements [5] that are qualified according to how their elements are supposed to participate in a relationship. For example, for *distributive* sets *all* ($\forall$) elements participate in the relationship and for *disjunctive* sets *at least one* ($\exists$) element of the set does. This provides an elegant way of expressing architectural relationships over sets of elements.

A conformance checking algorithm was developed using an approach which closely matches the above [3]. It was implemented in a PROLOG-like language with access to a Smalltalk code repository. An experiment was conducted to check conformance of the Smalltalk implementation

of some rule-based system, to a rule-based interpreter architecture, which is a specialization of the interpreter style. Because the components in this architecture mapped to sets of source-code artifacts, set qualifiers ∀ and ∃ were used to indicate the intended interpretation of these sets, as illustrated next.



Checking conformance was achieved by evaluating a logic query which is constructed from the architectural graph as follows. The concepts are mapped to sets of source-code artifacts. For example, 'Rule Interpreter' is mapped to the set of all Smalltalk methods that implement the interpretation of logic clauses. The conceptual relations are mapped to predicates representing high-level source-code dependencies. For example, 'Asks' is mapped to a predicate which checks whether a certain method invokes another one and actually uses the result afterwards. The edges and annotated set qualifiers state how these predicates are to be applied over the set elements.

For example, the relationships between 'Rule Interpreter' and 'Working Memory' are translated to logic queries that verify whether *every* rule interpreter method uses *some* data of the working memory, but only *some* of them update this data. The conjunction of all such queries yields the eventual query that checks architectural conformance.

Conclusion

The architecture is an important piece of knowledge about a software system. As such, we take the position that it is worth looking at existing knowledge representation formalisms to explore how they can be used or adapted to model different aspects of software architectures, capitalizing on existing notations, results, and tools.

To illustrate our position, we used conceptual graphs due to their pictorial and bipartite nature which is naturally suited to software architectures. Obviously, we only skimmed the surface. Conceptual graph theory includes other kinds of graphs, e.g., a data-flow graph represents computations, a schema represents background knowledge for a given type, a prototype represents a typical case. Schemata might be useful for domain-specific architectures, while prototypes might help in representing the non-variable part of a product-line architecture.

Different kinds of reasoning (besides classical logic) might also prove to be valuable for software architecture. For example, non-monotonic reasoning and belief revision might be needed to analyze all the implications of an envisaged architectural evolution and to explore different evolution paths.

In summary, we believe that applying knowledge representation techniques to software architectures will improve what architectural aspects we can represent, how we represent them, and how we manipulate them.

References

[1] D. Deridder and B. Wouters. The use of ontologies as a backbone for software engineering tools. In *Proceedings of the Fourth Australian Knowledge Acquisition Workshop AKAW99*, 1999. December 5-6, Sydney, Australia.

[2] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

[3] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *TOOLS 29 — Technology of Object-Oriented Languages and Systems*, pages 33–45. IEEE Computer Society Press, 1999.

[4] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[5] J. F. Sowa. *Conceptual Structures — Information processing in mind and machine*. Addison-Wesley, 1984.

[6] M. Wermelinger. A different perspective on canonicity. In *Proceedings of the Fifth International Conference on Conceptual Structures*, volume 1257 of *Lecture Notes in Artificial Intelligence*, pages 110–124. Springer-Verlag, 1997.