Modal Logic as a Design Notation

Areces, Carlos Felder, Miguel Hirsch, Dan Yankelevich, Daniel
Departamento de Computación
Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires

Abstract

*A notation to describe software system designs is given together with the means to verify properties over them. Designs are considered as models of a modal logic. The procedure to derive the modal model associated to a design, the algorithm to check properties over a model, the method to define new relations and the method of model filtration are presented. The proposed logic (*KPI *a poly-modal logic with inverse operators) is used as a property specification language verified through a model checking algorithm. The methods provided proved to be effective and simple to implement. A prototype tool has been developed in SML-NJ covering all functionalities described.*

1 Introduction

The successful development of software systems requires adequate tools. These tools have to be at the same time powerful, simple and formal. There are different tools for the different steps of the Software Development Cycle. In our work, we devote special attention to the Design Step.

Structured Design was the first technique reflecting and handling the complexity of modern systems. The techniques of Structured Design established a standard (graphic) language for the description of designs but it still lacked a *formal semantics* [11], and the formal proof of properties was still impossible.

Many researchers investigated this issue: [6] provides tools to obtain simplified views of the design, [1] studies specification of high level logic programs using operations on sets, [10] defines a query algebra for reverse engineering and [9] proposes a methodology to obtain from a high level specification of the system, a module decomposition scheme and refinement method based on an extended first order logic. We insist that a jump towards a higher level of abstraction is needed. In addition to techniques that provide simplified views of the system, we should be able to specify properties in a formal language and reason abstractly over the designs. This high level of abstraction is obtained when we consider designs as modal structures, the modal language giving the expressive power needed to formalize and verify properties of design.

Modal logics are widely used in verification of code and algorithms (PTL, Propositional Temporal Logic [12]; CTL, Computational Tree Logic [5], etc.) The original approach in this article is to carry this idea further and use descriptive logics (like modal logics) even during the Design step. We know of only another approach similar to ours ([4], a modal language proposed to describe system configurations) but our aim is more general.

We propose the logic system KPI as a property specification language. We prove that it is (logically) adequate for the class of models we want to describe, give then the logical equivalent to the basic Software Engineering operations on designs and exemplify its use in proving properties over a classical design example.

2 Modal Logic as a Design Notation

Modal Logics and Design The main idea of our work is that designs are graphs and Modal Logics are fine tuned formal languages to describe graphs (with the important property of being decidable.)

The design graphs will be the models of our logic: labeled transition system plus a valuation defining properties of the nodes. The transitions $-a \rightarrow$ are the relations existing among the different component of a design, the valuation establishes properties of the components. In this article we mainly consider the most simple property of a component, its name.

The main operators of our logic are $[a]$ and $\langle a \rangle$, similar to the universal and existential quantifiers of First Order Logic. $[a]\varphi$ is true of a component c in a design $\mathcal{D}$ iff the property φ is true in all a-related components c' $((\forall c')\,(c -a \rightarrow c' \Rightarrow \varphi$ holds in $\mathcal{D}))$. $\langle a \rangle \varphi$ is defined dually as $\neg[a]\neg\varphi$.

Poly-modal Logic with Inverse Operators Given the kind of graphs we should be able to describe (designs) and how they are used in Software Engineering we have to chose the adequate Modal Logic.

On one hand designers employ different relations to describe the interaction among components. Thus, we need different operators $[a]$, one for each of these

relations. On the other hand when a module m_1 is related to a module m_2 it is always understood that m_2 is also related to m_1 (if m_1 *invokes* m_2 then m_2 *is invoked by* m_1.) To account for this we will consider for each operator $[a]$, the *inverse operator* $[a]_i$ defined over the converse of $-a\rightarrow$.

The poly-modal logic with inverse operators KPI adds both extensions. An axiomatization, and a completeness and decidability result are established in [3].

We begin now to develop the methods we will use in the verification of design properties.

Verifying Properties on the Model If we consider graphs as models, we can use a model checking algorithm to decide the validity of KPI formulas. The model checker presented in the paper is a modification of the one in [5]. To determine the validity of a formula φ in a design $\mathcal{D}$, the algorithm will proceed in stages working recursively in the size of the subformulas of φ. The complexity of this algorithm is $O(t.\max(n, n.e))$ where t is the size of the formula φ, n is the number of components in $\mathcal{D}$ and e is the number of edges.

Example*: In our article we present a decomposition of the KWIC system (Key Word in Context) based in abstract data types (ADTs.) [7]. Starting from the model corresponding to the design of the KWIC, we automatically verify the property "An ADT uses only services from others ADTs". This property is represented by the formula* $\varphi = \langle//\rangle_i\top \rightarrow [/]\langle//\rangle\top$ *which is formally proved to hold in all nodes of the graph.*

Modifying the Model The verification of a design is commonly associated to the proof of properties about the involved relations. Given the usual complexity of designs, we want to obtain a global view defining high-level relations or to pay attention only to certain subset of them. In a modal language, accessibility relations are associated to the modal operators of the logic. The equivalent to create or eliminate accessibility relations would be the definition of a new modal language with more or less modalities.

Model Expansion. What we want to do is to introduce new modalities as a kind of *shorthands* in the language, with their associated relations appearing explicitly in the new model. This new high-level relations will have their behaviors determined by the basic relations that take part in the definition. Our aim is similar to the ideas in Beth's Theory of Explicit Definition for first order logic which describes how new symbols can be defined without adding expressive power.

Given that our interest is to characterize syntactically the construction of new relations from basic relations, it is enough to give definitions for the operators we will use. A formula defining each of the basic operations (union, composition, etc.) is provided and we prove that each model ($\mathcal{D}$) can be expanded to a new model ($\mathcal{D}^+$) where the meaning of all formulas is preserved and the new operators are interpreted according to their intended meaning.

These characterization results are obviously limited by the expressive power of the logic. For example, it is impossible to characterize in KPI the complement operation. Other operations over relations (as intersection and subtraction) cannot be fully characterized over the class of all models, but they can be completely captured in the class of the models that correspond to designs.

Example*: We define from the basic relations* invokes *and* is_part_of *in the KWIC, the relation* uses_ADT = invokes $\circ$ is_part_of. *The design obtained* $\mathcal{D}^+$ *is identical to the original one with the addition of the relation* $\rightarrow$ *corresponding to* uses_ADT. *We prove that,* $\mathcal{D}^+$ *use* $\langle\ \rangle$ *as shorthand of* $\langle/\rangle\langle//\rangle$ *providing the explicit relation corresponding to* $\langle\ \rangle$.

Model Abstraction. Moving in the other direction, we want to obtain simpler models providing us with a clearer picture of the design. When proving a given property, only some parts of the model are relevant and we would like to get rid of the rest.

If $\mathcal{P}$ is a set of properties (formulas), a $\mathcal{P}$-filtration builds a *collapse* of the original model preserving the validity of the formulas that take part in the construction. From the point of view of Design, the method of filtration defines a new design that acts as an abstraction of the original. This new model shows a simplified view of the design in which the non interesting details have been eliminated while new, more general and abstract concepts appear.

A result of preservation under filtration shows how the existence of a $\mathcal{P}$-filtration forces formulas in $\mathcal{P}$ to be "treated equivalently" in both models. The filtered model shows the design as it is "seen" from the point of view of the properties that were chosen as relevant and included in the filtration set $\mathcal{P}$. The model obtained by the filtration depends intrinsically of the set of properties $\mathcal{P}$ involved. A well defined $\mathcal{P}$ returns the expected abstraction.

The filtration algorithm has complexity $O(t.\ \max(n, n.e)\ +n^2)$, where t is the size of $\bigwedge\mathcal{P}$, n the number of nodes and e the total number of edges.

Example: *From the KWIC design and the formula* $\langle//\rangle_i\top \rightarrow [/]\langle//\rangle\top$ *used in the model checking example we obtain the filtered model. The modules of the original model are automatically brought together in accordance to the role they play with respect to this formula. The classes* ADTs, Function, Input_Output *and* Mas-

ter_Control *are defined and their interplay clearly outlined in the abstraction. See [3] for details. This abstraction not only shows that ADTs use functions that are part of others ADTs, but also shows the design architecture of the system.*

3 Conclusions

From the similarities among the models used in Modal Logic and Software Design a logic was defined to be used as a specification language in the verification and description of designs. The design graphs are considered models of the logic KPI. The methods used to verify properties are developed from techniques in classical modal logic. We provided the procedures to derive the modal model associated to a design, the model checking algorithm to verify properties, the expansion method to define new relations and the model filtration method for abstraction. We believe that these methods help to accomplish the tasks that are commonly found in the process of Software Engineering and to formalize the process of design verification. The key fact is that given the exact definition of the meaning of a formula provided by the logic KPI the correctness of these methods can be formally proved.

The simple application examples we study show that the methods proposed are useful and they can be automated. A prototype tool covering all the concepts and functionalities described was implemented using the language SML-NJ [3]. But the methodology was yet not used over a real-world design problem.

The aim of this work was to set a strong and basic background for the use of modal techniques in Design. Several possibilities of research in Modal Logic applied to Software Engineering are open: The study of the notion of model expansion must be completed. The results obtained through the filtration method are encouraging but the real power of this technique as an abstraction method remains to be studied in detail. The application examples in the paper shown how to handle in our formalisms graphs with complex information in the modules. We do not claim though, that this can be done in all cases. How should the formalism be modified to cover other types of information? Finally, the idea need not be restricted to Design. The approach might also be useful in the description of software architectures, even though there the problem has higher complexity. The graphs used in this field seem to capture correctly the static characteristics of the system, but not the dynamic that defines the interaction among them [7, 2].

Acknowledgments This work was partially supported by UBACyT EX-186 and ECC KIT#125 projects. Areces is under a grant from the British Council (#ARG0100049.)

References

[1] Agustí, J., Robertson, D., and Puigsegur, J. Grasp: A graphical specification language for the preliminary specification of logic programs. Technical report, Institut d'Investigació en Intel.ligència Artificial (CSIC), 1995.

[2] Allen, R. and Garlan, D. Formalizing architectural connection. In *International Conference on Software Engineering*, May 1994.

[3] Areces, C. and Hirsch, D. Modal logic as a software engineering tool. TR 96-0004, Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales. Dep. de Computación., 1996. http://www.dc.uba.ar.

[4] Bernhard, S. and Tiziana, M. Automatic synthesis of design plans in METAframe. Technical Report MIP-9610, Universität Passau. Fakultät für Mathematik und Informatik, 1996.

[5] Clarke, E., Emerson, E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[6] Cosens, M., Mendelzon, A., and Ryman, A. Visualizing and quering software structures. In *ICSE'92 Proceedings of the 14th International Conference on Software Engineering*, 1992.

[7] Garlan, D. and Shaw, M. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, I, 1993.

[8] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.

[9] Maibaum, T., Sadler, M., and Veloso, P. Logical specification and implementation. *Lecture Notes in Computer Science*, (18), 1984.

[10] Paul, S. and Prakash, A. A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3), March 1995.

[11] Perry, D. Software interconnection models. In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–69, Monterey, California, March 1987.

[12] Pnueli, A. The temporal logic of programs. In *In Proceedings of the 18th. Annual Symposium on the Foundations of Computer Science*, New York, 1977. IEEE.