14TH WORLD CONGRESS
at Georgia Tech

on Computational and Applied Mathematics

July 11-15, 1994

School of Mathematics
Georgia Institute of Technology
Atlanta, Georgia
USA

Proceedings in 3 Volumes
Volume 3

IMACS '94

Proceedings of the 14th IMACS World Congress on Computation and Applied Mathematics

July 11-15, 1994, Georgia Institute of Technology, Atlanta, Georgia, USA

in three volumes

VOLUME 3

Numerical Methods and Analysis
Finite Differences and Elements
Differential Equations
Intervals

EDITED BY: W.F. AMES
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA, USA

On Knowledge Base Refinement[1]

Carles Sierra and Lluís Godo
Institut d'Investigació en Intel·ligència Artificial (IIIA)
Spanish Council for Scientific Research (CSIC)
17300, Blanes, Catalonia, Spain

1 Motivation

Incremental programming is considered a safe way of getting correct programs. The programming methodology consists on defining small pieces of code that are later on ellaborated, combined, or redefined, up to a point in which the final program is obtained. In some languages this is materialized in a three-step process for each piece of the program: definition of an initial specification, definition of an implementation of it, and finaly, verification of the correctness of the implementation. The combination of these verified pieces gives the final complex program. This is the way, for example, SML [Milner et al, 1990] works. In Knowledge Based Systems the methodology seems also useful despite the fact that precise specifications are, in many cases, difficult to obtain. The idea of working with small pieces of knowledge has been increasingly used in the artificial intelligence field. Objects, agents or modules are the components from which bigger systems are defined. However, no clear correlate of the incremental programming methodology as stated before has been applied. In this paper we present some initial ideas in this direction presented in terms of a modular language, from which we hide the module components details. A set of knowledge units will define a module, and the methodology will consist on considering knowledge and specification as interchangeable terms. This is done so because usualy knowledge units are nothing but restrictions over the relation between factual data and results, as specifications do. To cope with the idea of ill-defined problems (problems with partial, incomplete or even erroneous specifications) we propose a n-step process in which each module (specification) is implemented using another module (new spefification) which is again implemented using a module, and so on, up to a point in which the knowledge in the final module is considered to be competent with the problem to solve. Every step is verified as in the classical process. This incremental process is modelled by means of an operation called *refinement* from which some formal properties are outlined in the paper.

2 Basic Definitions

We will concentrate in the view of a module as a black box which relates inputs with outputs.

Definition 2.1 Given a set of fact identifiers F, the set of *interface signatures*, namely Σ^F, is defined as $\Sigma^F = 2^F \times 2^F$. A signature σ is a pair (I, O) where $I, O \subset F$.The first component of a signature is the set of *input facts*, and the second componenet is the set ot *output facts*. Given a particular module signature σ = (I, O) the next accessing functions are also defined, $In(\sigma) = I$ and $Out(\sigma) = O$. The inclusion relationship in Σ^F is defined as $\sigma_1 \subseteq \sigma_2$ *iff* $In(\sigma_1) \subseteq In(\sigma_2)$ *and* $Out(\sigma_1) \subseteq Out(\sigma_2)$.

Definition 2.2 A *code* C for a given domain of values *Val* and a signature σ, is any function $C: Val^{In(\sigma)} \rightarrow Val^{Out(\sigma)}$ that relates mappings of input facts of the signature into mappings of output facts of the same signature. That is, if $h: In(\sigma) \rightarrow Val$ then $C(h): Out(\sigma) \rightarrow Val$

Finaly, the abstract notion of module we consider is modelled as a pair defining interface and knowledge. From now on the set of fact identifiers F and the set of values *Val* are considered fixed.

Definition 2.3 A module m is a pair $m = (\sigma, C)$ where C is a code for the signature σ or $m = \perp$. The set of modules on F and *Val* will be denoted by Σ_M.

---

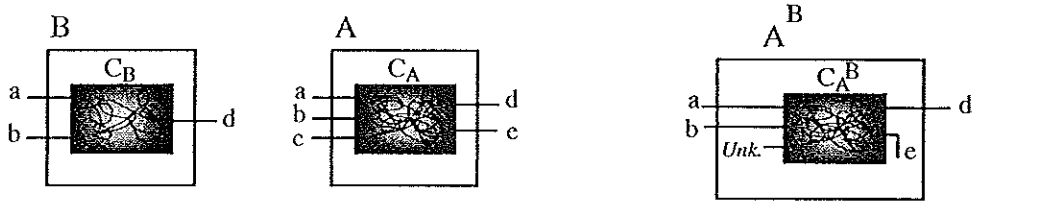
[1]This research has been partialy supported by the CICYT project ARREL (TIC92-0579-c02-01)

We want to study the behaviour of modules, that is, which outputs do they produce depending on the inputs. The mimetization of modules tries to see how a module must be modified to make it compatible with the interface of another module.

Definition 2.4 The *mimetization* of a module $B=(\sigma_B, C_{\sigma_B})$ by a module $A=(\sigma_A, C_{\sigma_A})$, is a new module noted A^B constructed by transforming A in the next way.

$$A^B=\begin{cases}(\sigma_B, C_{A^B}), & if A\neq\perp, B\neq\perp, \text{ and } \sigma_A\supseteq\sigma_B\\ \perp, & otherwise\end{cases}$$

$$\text{with } C_{A^B}(h)=\left[C_{\sigma_A}(h^B)\right]_{Out(\sigma_B)} \text{ and } h^B(g)=\begin{cases}h(g), & if g\in In(\sigma_B)\\ unknown, & otherwise\end{cases}$$



Module B Module A Mimetization of B by module A

The central idea in what concerns verification in our approach is that when we refine a module we want to get a new module with the same interface but with "more precise" knowledge. We will understand "precision" with respect to a pre-defined ordering relation over the *Val* set. This partial ordering in *Val* will be noted by $\lhd$ henceforth.

Definition 2.5 We say that a module $A=(\sigma_A, C_A)$ has a code C_A with a *more precise behaviour* than the code C_B of a module $B=(\sigma_B, C_B)$ such that $\sigma_B\subseteq\sigma_A$ and noted $C_A\prec C_B$ or simply $A\prec B$, if and only if it holds $[C_A(h)](f)\lhd[C_B(h)](f)$ for all $h\in Val^{In_{\sigma_B}}$ and for all $f\in Out_{\sigma_B}$.

In general the verification of being more precise is computationaly untractable: it is equivalent to the SAT problem. In some cases the complexity can be reduced when the knowledge units are written in a simple way, as it is the case for the architecture used as example in the next section. Finaly we propose a definition of refinement.

Definition 2.6 The module A^B is called the *refinement* of a module B by A if and only if the behaviour of the code of A^B is more precise than the code of B. Associated to this relation we define the function ":" as

$$A:B=\begin{cases}A^B, & if C_{A^B}\prec C_B\\ \perp, & Otherwise\end{cases}$$

Equivalently, a module A is an *expansion* of a module B, or a module B is a *contraction* of a module A, if and only if the behaviour of the code of A^B is more precise than the code of B. Operationaly we have the following functions:

$$A>B=\begin{cases}A, & if A:B=A^B\\ \perp, & Otherwise\end{cases}\qquad A<B=\begin{cases}A, & if B:A=B^A\\ \perp, & Otherwise\end{cases}$$

Now we present, based on definitions in the previous section, some properties of the refinement operation.

Proposition 2.7 For any modules A, B and C next properties hold:

1. *if* $B^\alpha\neq\perp$ *then* $(C_A\prec C_B\Rightarrow C_{A^\alpha}\prec C_{B^\alpha})$
2. $h^{C^B}(g)=h^{B^C}(g)$
3. *if* $B:C=B^C$ *then* $(A^B)^C=A^{B^C}$
4. *if* $A:B=A^B$ *and* $B:C=B^C$ *then* $A:C=A^C$

Corollary 2.8 The refinement-related relation $\prec$ on the set of modules Σ_M (see definition 2.5) is a partial ordering, i.e. $\prec$ is reflexive, anti-symmetric and transitive.

Corollary 2.9. The structure $(\Sigma_M, >)$ satisfies the next properties which are the operational counterpart of corollary 2.8:

1. $A > A = A$
2. $A > B$ *and* $B > A$ *implies* $A = B$
3. $(A > B) > C = A > (B > C)$

Corollary 2.10 The structure $(\Sigma_M, :, >)$ satisfies the next properties.

1. $A : \bot = \bot : A = \bot$
2. $A : A = A$
3. *if* $A:B = A^B$ *and* $B:C = B^C$ *then* $(A:B):C = A:(B:C)$

It is worth noticing that ":" and ">" are not commutative operations, as corollary 2.8 entails. Also we note that in general the refinement operation ":" is not associative unless corollary 2.10-3 shows a restricted form of associativity.

3 Application to MILORD II

MILORD II is an architecture based on modules, many-valued logics and reflection [Sierra and Godo, 1993]. We will particularize the definitions of the second section here.

Definition 3.1 (Val in MILORD II) *Val* is the set of intervals defined over a finite set of uncertainty linguistic values $L_n = \{a_1, \ldots, a_n\}$ with a total order relation defined on it $a_1 < a_2 < \cdots < a_n$. The partial order relation $\lhd$ on *Val* is then the usual interval inclusion, that is, $\forall v_i, v_j \in Val . v_i \lhd v_j \Leftrightarrow v_i \subseteq v_j$

Definition 3.2 (Code in MILORD II) A code C of a MILORD II module $m = (\sigma, C)$ is implemented as a set of rules of the next type $C = \{(P_1 \wedge P_2 \wedge \cdots \wedge P_n \rightarrow Q, V) \mid P_i, Q \in F, V \in Val\}$, where the set of fact identifiers F is a set of propositional variables.

The interpretation of these rules is logic-oriented and explained in [Sierra and Godo, 1993]. Considering the axiomatics of the many-valued logic used, it is possible to flatten all these rules to get an expression for the value of output facts. Namely, $C(h)(fact) = \bigcap_{i=1}^{n} (h(P_{i_1}) * \cdots * h(P_{i_m}) * V_{i_1} * \cdots * V_{i_m})$ where C is the code function of module m, h is a given interpretation for the input facts, *fact* is an otput fact, P_{i_j} are input facts, V_{i_j} are elements in *Val* corresponding to the rules of the code, and * is a t-norm like operator used in module m to propagate the uncertainty values. By using the characteristics of intersection and the above mentioned t-norms operators, it is possible to obtain an algorithm to check whether $C_A \prec C_B$ holds that shows a reasonable efficiency in average. The worst case complexity is, however, the same as that explained in the second section.

4 Conclusions and Further work

We have presented in this paper some ideas about the incremental programming methodology in Knowledge Based systems through the notion of *module refinement*. Some interesting properties of this refinement operation are outlined. A brief description of how this methodolgy is used in the MILORD II architecture is also presented. Future work will consist in extending the definitions and properties to consider the refinement of declarative control and reflection techniques, common to many present Knowledge Based architectures.

References

Milner R., Tofte M., Harper R. (1990): *The definition of Standard ML*, MIT press.

Sierra C., Godo L. (1993): "Specifiying Simple Schedulling Tasks in a Reflective and Modular Architecture", in: Treur J. and Th. Wetter (eds.) Specification of Complex Reasoning Systems, Ellis Horwood pp. 199-232.