INTEGRATING DECLARATIVE KNOWLEDGe PROGRAMMING STYLES AND TOOLS IN A STRUCTURED OBJECT AI ENVIRONMENT

Hihai Barbuceanu Stefan Trausan-Matu Balint Molnar

Institute for Computers and Informatics
8-10 Miciurin, 71316 Bucharest 1
phone 653390. telex 11891 lcpci r
RUMANIA

Central Research Institute for Physics
H-152S Budapest 114, P.O.B. 49
HUNGARY

ABSTRACT

XRL is an integrated knowledge programming environment whose major research theme is the investigation of declarative knowledge programming styles and tools, and of the way they can be effectively inlegrated and used to support AI programming. This investigation is carried out in the context of the structured-object representation paradigm which provides the glue keeping XRL components together. The paper describes several declarative programming styles and associated support tools currently available in XRL.

I. INTRODUCTION

Recent research [1] prompts the view that the success of AI knowledge based programs is a direct consequence of a specific approach to programming in which the programmer constructs declarative specifications directly interpretable by the supporting programming system, rather than of a different nature of AI programs.

If one shares this view (as we do), then developing better declarative programming languages and language features providing leverage for this approach to programming becomes an essential research direction.

The XRL system reported in this paper is one such attempt. It is an integrated knowledge programming environment whose major research theme is the investigation of declarative knowledge programming styles and features and of the way they can be effectively integrated and used to support AI programming.

The paper presents the most important declarative programming styles and associated tools currently available in XRL. These tools are organized in the layered architecture shown in fig.1.

The structured object layer forms the substrate of the whole environment providing the glue which keeps everything together. The instantiation framework supports a generalized view of the ubiquitous frame instantiation process. The description based programming system provides a novel declarative programming style which embedds a mathematical oriented description language in the structured object environment. The semantics oriented programming framework offers a specific semantic construct based approach to programming which supports maintainance and evolution. Finally, the self generation tool applies this approach to XRL itself.

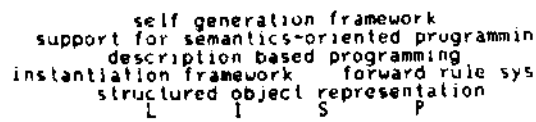

Fig. 1: XRL architecture

II. STRUCTURED OBJECT SUBSTRATE

A. Structured objects

This layer provides an object-centered programming system [2-6]. Structured objects come in two forms: generic objects, called units which stand for classes of objects and individual objects (instances). Units are composed of slots which have values and meta-descriptions. Meta-descriptions, which also apply for units as a whole, are specified in terms of other first order units which can be meta-described themselves by means of other meta-units a.s.o..

A unit example is shown in fig.2. Its meta-description appears in the self slot. The notation (a <unit name>...) defines a "virtual copy" of <unit name>. It is an anonymous unit which can be used in a single place. It inherits all the slots defined by <unit name>. The slots mentioned in the notation are added to the new unit, overriding the corresponding ones from <unit name>. At this level meta-descriptions are used to store specifications of inheritance relations (the supers slot) and the association of methods to selectors (the draw and refine-by slots). A full language for specifying method combination is also provided in [7].

```
[House
 self (a HouseMetaUnit
        supers (Building Property)
        subtask-processing-order sequence
        pre-cond (ask "Working on House?")
        post-cond (TrueSlot is-affordable?)
        not-affordable (an Event
                           slot is-affordable?
                           predicate not
                           handler HouseNotAfdb)
        draw DrawHouse)
 room-types (setof(set Bedroom Living Dining
                       Bathroom Kitchen)
                   such-that UserApproves)
            (a SODLMetaUnit refine-by Filter)
 rooms (setof ;O: (a :1: room))
       (a SODLMetaUnit
         labels (:1: :O:)
         :1: (a SpecializationMultiSetConstr
               specializations (any room-types)
               multiplicity AskUser)
         :O: (a WholeSet))
 is-affordable? (using(income(path owner income))
                   (HouseAfdb? income cost
                               maintainance-cost))
 maintainance-cost (ComputeMC rooms)
 cost (a Cost)
      (an ADTValuedSlot)]
```

Fig. 2: The House unit

This layer also supports an abstract data type (ADT) programming facility. ADT-s are described as usual units (slot cost in fig.2), but are interpreted by a special compiler for variable creation and manipulation (somewhat similar ADT-s are provided in [5]).

As efficiency at this level is vital for the whole environment, time consuming processes such as method combination, message sending and ADT creation and manipulation are carefully compiled.

III. THE INSTANTIATION FRAMEWORK

A powerful and widely used mechanism for bringing the knowledge encoded in structured-object representations to bear is instantiation, a process by which the knowledge contained in a Generic object is applied to the construction of a similarly structured terminal object. In spite of the ubiquity of instantion in frame or object-centered systems, few languages explicitly support this process. One of these rare cases is LOOPS [33] with its composite objects, but even here only a rigid recursive mechanism is provided. XRL fills this gap with a generalized instantiation engine discussed below.

A. Instantiation tasks

Central to the instantiation framework is the concept of task. The task is the processing unit of the instantiation system. Tasks may represent ?eneric objects in course of instantiation, AOT-s ?o be created or operated upon or arbitrary MSP expressions to be evaluated. Each of this is allocated an instantiation region on the blackboard. Considering the House unit (fig.2), the system will associate a task to the unit as a whole and several subtasks to each of its slots. Among the latter we note an ADT to be created (the cost slot) and a LISP expression to be evaluated (the maintainance-cost slot). The tasks generated for room-types and rooms belong to the description-based programming layer to be discussed later on. As illustrated by the example, there exists a task-subtask hierarchy stemming from the structured nature of units and inducing a tree-structured organization of agendas.

B. Specifying task processing

Task processing is accomplished according to an instantiation protocol. This protocol is defined as a convention about the meaning of a number of selectors involved in message passing activities triggered by the various components of the instantiation system.

The instantiation protocol specifies the following issues: (1) The conditions under which a slot is to be processed as a subtask. (2) The pre-conditions enabling a task to be selected for execution. (3) The post-conditions for successful task processing. (4) Colateral actions to be performed before or after task processing. (5) Task priority; a given task may decide on its own priority as well as on the priorities of its subtasks or of other tasks whose results are needed for its own processing. (6) Actions to be carried out for executing a task. (7) Descriptions of exceptional events and of the associated event handling activities.

Some of this possibilities are illustrated in fig.2, the self slot.

C. Communication and synchronization

In order to allow tasks to communicate to each other and to provide mechanisms for programming task placement in the agenda, the system introduces two new primitives. The path primitive (see slot is-affordable? from fig.2) specifies dynamic access paths to the instantiation regions of other tasks. Thus, (path owner income) creates an access link to the income slot of the instantiation region associated to the unit filling the owner slot of House unit . Path-s automatically schedule the tasks using them after the value they need becomes available.

The second primitive is the using construct (see also slot Is-affordable? in fig.2) which binds variables to the objects referenced by its inner paths. In fig.2 variable income will receive the value of the owner's income (when available).

To program various control policies, the message passing protocol allows (I) any task to modify its own orir.rity, (?) a task to modify the priority of the tasks it references through path-s.

D. System organization

Task processing is organized into "instantiation systems" (IS) which are declaratively described as units (fig.3).

```
[HouseDesignIS
 self (an ISMetaUnit
        supers (InstantiationSystem))
 spaces (specs-acq design)
 specs-acq (a Space
              unit House
              task-sel-fn task-eligible?
              trace-on T)
 design (a Space
           unit HouseDesign
           pre-cond (done-space specs-acq)
           task-activate-fn task-activate)
 states (s1)
 s1 (an ISState
       spaces (specs-acq design)
       select-space first-ok
       interrupt-space no
       finish-space no-tasks
       next-state *final-state*)
 syst-pre-cond T
 to-run-fn ISRun)
```

Fig. 3: Instantiation system for house design

An IS is composed of an agenda and a number of spaces (blackboard zones). In each space a network of instantiation regions generated by the processing of an initial unit can be constructed. Each IS may be individually manipulated by the programmer. This means it can be created, run, interrupted, resumed, modified. This mechanism offers flexibility in dividing the initial problem into subproblems and in handling the interactions between them. For example, we used it in a number of design applications to manage different phases of the process such as specification, preliminary design, detailed design, evaluation [8].

The system also provides a dependency handling facility. Dependencies are created by explicit paths or by referencing slot names in the methods or expressions used in a unit (see fig.2).

F. Control regimes

Besides instantiation, other control regimes *are* available at this level. Dependency driven backtracking exploits the dependency records allowing selective network modification by tracing back the elements from which the offending value was derived.

The replay regime allows reprocessing a network modified by dependency driven backtracking by reinstalling "seed" tasks in the agenda and reusing previously derived results when processing these tasks.

The event handling regime, entered whenever an event is generated, offers an encompassing set of event handling options. In fig. 9 some of the slots and units involved in declaring events and specifying handlers *Are* exemplified. The not-affordable event is raised by the violation of the task post-condition and is treated by the House Not Afdb routine.

It is now possible to understand how the House unit will be treated if passed to this problem solver. All its slots (including those inherited) will be considered as subtasks and sequentially processed. The unit will not be processed unless thp user approves it. Because of the existing dependency relations the is-affordable? subtask will be scheduled after the owner's income, cost and maintainance-cost tasks.

IV. DESCRIPTION BASED PROGRAMMING

The PBP layer is an attempt to formalize a notion of higher level (HL) structured object and to create a programming system based on it.

Programs written in terms of HL structured

objects and their specific inference models can be factored into a description of the specification (what should the program compute) and a description of the implementation (how should the computation proceed). These two descriptions are separate, thus allowing separate understanding and manipulation of specifications and implementations.

We view three major advantages of this style of programming. First, the higher level character which allows programs to be specified in abstract, mathematical terms for which a clearly understood semantics exists. Second, the separation of specification from implementation which makes it possible to understand the specification of a program without having first to simulate in mind its implementation, which is what necessarily happens with procedural representations and imperative programming in general. Third, the introduction of a more mathematical view on structured objects which has a positive influence on the understanding and utilization of this representation paradigm.

A. The set oriented description language (SODL)

While the meaning of "frames" or structured objects varies heavily across the range of existing languages, SODL builds upon the interpretation of an XRL generic object as a class of terminal objects (instances). This is consistent with the stand taken by the underlying instantiation layer which constructs instances of generic objects. SODL provides a set oriented language for describing the classes of objects which can fill the slots in instances of generic objects.

First, here is the syntax of SODL.

```
Description -> Unit
               (oneof Set [suchthat P])
               (setof Set [suchthat P])
               Set
Set -> Unit
       (setof Set [suchthat P])
       (or Set ...  ) ; set union
       (and Set ...  ) ; set intersection
       (diff Set ...  ) ; set difference
       (set Elm ...  ) ; explicit enumeration
Elm -> Lisp-object ! Instance
```

SODL descriptions can be divided into two categories. Element descriptions (oneof Elm) describe an entity as belonging to a set of objects. Set descriptions describe sets (1) as sets of instances of a given unit, (2) as subsets of another set (setof), and (3) by usual set operations. In the current version, predicates are lambda expressions of one variable.

In fig.2, two very simple descriptions are used. In the room-types slot, a setof description describes the possible room types in the house. The setof description in the rooms slot specifies that it can be filled with a set of instances of the Room unit. As a more complex example,

```
(oneof(setof(a NiceGirl studies CS)
             suchthat (height < 180))
       suchthat (eyes-colour = blue))
```

describes a girl with blue eyes from a set of NiceGirl-s studying AI and having height < 180.

B. Semantics of SODL

A SODL description which fills a slot in a generic unit describes the class of individual objects or sets of objects which can possibly fill that slot in any instance of the unit. A SODL description thus stands for a class of individuals or a class of sets of individuals.

To clarify the meaning of SODL, we briefly define (in a manner akin to [9]) a formal semntics for SODL based on the notion of description extension. The extension of a description d is the set of elements described by d. A function E from descriptions to either elements or sets from any domain D is an extension function if and only if it satisfies the following properties:

1.E[Unit]=Powerset(Instance-of(Unit))
2.E[(oneof S P)]={x/x∈I&I∈E(S)&P(x)}
3.E[(setof S P)]=Powerset{x/x∈I&I∈E(S)&P(x)}
4.E[(or S1...Sn)]={I/I=UIi&Ii∈E(Si)}
5.E[(and S1...Sn)]={I/I=∩Ii&Ii∈E(Si)}
6.E[(diff S0 S1...Sn)]={I/∀x∈I0&I0∈E(S0)->
                        ->x∉Ii&Ii∈F(Si)}
7.E[(set E1...En)]={(E1...En)}
8.F[Lisp-object!Instance]={Lisp-object!Instance}.

Looking at how a subset of description's extension is completed, it is important to analize the first three rules above. All of them describe a kind of choice from alternatives. The oneof rule describes the choice of an element. The setof rule describes the choice of a subset of a set while the unit rule describes the choice of a subset of instances of a given unit.

C. Reasoning with descriptions

An important notion at least two useful reasoning models can be based on is that of refining a description. If d1 and d2 are descriptions, then d1 is a refinement of d2 iff E(d1)⊂E(d2) for any E and D. Thus stated, refinement is the same thing as the subsumed by relation defined in [10]. The description language and the way in which refinement is used in reasoning differ however fundamentally from languages like KLONE [10] , OMEGA [12] or KRYPTON [1]. Namely, we adopt a generative reasoning model in which the problem solver starts with an initial description and incrementally produces more and more refined descriptions from it, until a satisfactory level of refinement is attained (including of course the determination of terminal elements). For example, in fig.4 an initial description and two stages in its incremental refinement process are illustrated.

```
[AIGirls                                  ; initial unit
 elements (setof (a WiseGirl))]

[AIGirls-05                               ; stage 1
 elements (setof(or(a Girl age 39)
                   (a Girl boyfriend (an AIGuy))
                   (a Girl IQ 200))]

[AIGirls-17                               ; stage 2
 elements (set Lucy Ann Jane Lola)]
```

Fig. 4: An example of incremental refinement

A second possible reasoning model is a recognition-based one in which a given unit is compared to other units in a knowledge base. The comparison may reveal (most specific) subsumers in which case the unit is classified in the knowledge base (in the style of the above cited languages [10-12]) or (most general) refinements in which case the unit may act as a template for associative retrieval. The recognition based mode can and should be integrated with the generative one. For example, the object Jane (fig. 4) might had been retrieved as a refinement of (a Girl IQ 200). Currently, the existing DBP system supports the generative refinement model and only rudimentary recognition mechanisms. Most important, DBP lacks for the moment a program able to decide if, given two SODL units, one is a refinement of another. We view integrating generative and classificatory reasoning as an important topic for the near future.

D. Operationalizing the incremental refinement model

1) The nature of operationalization. Consider the description:

```
(oneof :0: (setof :1: (a :2: Student studies AI)
                      suchthat (height > 180))
           suchthat (age < 30))
```

One possible operationalization is as follows. At :2: search the KB for instances for the given unit for at most 10 units of time. At :1: filter the resulting list with the given predicate and retain the first 5 instances. At :0: take the first instance satisfying the predicate.

Several aspects are worth noticing. First, an element from the extension is computed. Second, it is determined by search. Third, with the imposed resource limitations it may well happen that no suitable elements be found.

A lot of other operationalizations can be obtained by altering the above one: (a) replace search with instance construction (done with the instantiation processor); (b) instead of two filtering steps perform only one and-ing the two predicates (done through a syntactic transformation); (c) replace unit Student with the union of its specializations. Provided these specializations have more efficient methods for constructing or retrieving instances, this semantic transformation may improve the efficiency of step :2:.

2) The operationalization framework. The framework recognizes four dimensions along which operationalization can be performed: search, construction, syntactic and semantic description transformation. We have adopted a rule-based form for encoding the operationalization procedure which fall into the above categories. For this purpose we use the XRL production rule system. This a forward reasoning system which allows defining both rules and rule interpreters in a uniform and extensible object oriented notation (similar approaches are [4,15]).

```
[UnitToUnion
 self (a SemanticTrMeta)
 variables (Specs Unit)
 descr-conditions dc1
 proc-conditions (pc1 pc2)
 dc1 (IsUnit Unit)
 pc1 (Exists Specs (SpecOf Unit))
 pc2 (not(HaveMethods Unit '(toconstruct tofind)))
 recommend (HaveMethods Specs
                '(toconstruct-inst tofind-inst))
 new-description (MakeUnion Specs)]
```

```
If: (1) the description is a unit
    (2) there exist specializations of it
    (3) the units does not have to construct or
        to find instance methods
Then: transform it into a union of its
      specializations
Recommended: if the specializations have methods
             either to construct or to find
             instances.
```

Fig. 5 UnitToUnion semantic transformation

Fig.5 illustrates a semantic transformation rule which transforms a unit into the union of its specializations. Recommendations identify situations where the rule is likely to be especially effective.

The framework defines several standard operationalization regimes. One is interactive, user-guided operationalization in which user selects the rules and system applies them. Another is a default automatic one which attempts to operationalize any DB program even if no operationalization information is provided (done using defaults). Finally, the third executes a single rule for each description, covering the case when the programmer can precisely define the operationalization activities.

The processing of the descriptions used in fig.2 can now be understood. The room-types slot selects the types of rooms by filtering a given set of alternatives with a predicate which triggers user questioning. The rooms slot specifies that a set of specializations of unit Room must be constructed in the following manner. The specializations are those units whose names appear in the room-types slot and the number of occurences for each specialization (i.e. room type) is interactively given by the user (label :1:) and the whole set thus produced will be retained as the slots values (label :0:).

E. Applying DBP

We view the usefulness of DBP for applications which can be modelled by an incremental refinement process which starts with high level but operational specifications and incrementially transforms them into lower level descriptions.

Many design problems can be modelled in this way [14], ranging from mechanical design [15] and civil engineering design [16] to program synthesis [17]. Our use of DBP has been in this direction and is reported in [8,19].

If we consider the distinction between classification and constructive problem solving made in [18], DBP appeares as a tool for the latter. It is a mechanism for creating and assembling local solutions and objects into higher order aggregates which are not part of a pre-existing ennumeration. This may suggest DBP (with the languages below) as a generic tool for this type of problem-solving. To attain this objective, what we currently have can only be a first step. Much work is still required, especially more applications.

V. SEMANTICS-ORIENTED PROGRAMMING AND SELF-GENERATION

The programming tool presented in this section supports a semantics-oriented programming methodology whose major advantage is a knowledge-based mechanism supporting program modification. Other advantages are better understandability and reusability of the programs thus developed.

A. Semantics-oriented programming constructs

The central concept of our approach is the semantic construct notion. The nature of semantic constructs can be best understood by means of an analogy between programming language constructs and the constituents of grammars used in natural language understanding.

The constructs of general-purpose programming languages are domain-independent structures(e.g. if statements or real data types) in terms of which programs in any problem domain can be constructed. They are analogous to the constituents of general NL grammars (such as noun phrase or verb phrase). Semantic constructs play a role similar to the constituents of semantic grammars [20]. They are related to the semantics of the problem domain and provide a strong connection between this domain and the language the domain is encoded in.

An example may help elucidate this. Assume the domain of discourse is the instantiation framework of XRL and we have several routines whose processing depends on the state of the current task. In a general-purpose language like LISP we could encode these routines according to the following pattern:

```
(defun <name> (task)
 (prog(s)
  (cond((eq(setq s(caddddr task)) 'new)
        <processing the new state>)
       ((eq s 'active)
        <processing the active state>)
       ...
       (t (error "task state inexistent")))))
```

For this particular domain, we can also define a special semantic construct called, e.g. TaskProcessingProcedure). In a structured object notation the concept could be defined as shown in fig.6. Also in fig.6 we illustrate the general pattern for using it.

```
[TaskProcessingProc
  self (a SemanticProcedureMetaUnit
          supers (DataParameterizedProc))
  arguments (task)
  args-types (task)
  test task:state
  possible-values (new active ...done)
  error-signalig (error "task state inexistent") ]

(a TaskProcessingProc
   new <processing for the new state)
   active processing for the active state>

   done <processing for the done state))
```

Fig. 6: TaskProcessingProc definition and use

Comparing the unit for using the construct with the LISP code shown earlier we note the following clear advantages:

(1) The notation is easier to perceive and understand as things like the prog and cond struetures,the tests in each cond clause, the way of accessing the task state and the local variable do not appear. Their function is automatically provided by the semantic construct.

(2) Knowledge about the problem domain, such as the possible task states, is embedded in the definition of TaskProcessingProc and automatically applied when the construct Is uti1ized.

(3) Programming disciplines, such as the manner of error signaling, can be clearly enforced.

(4) Programming knowledge, e.g. the manner of accessing the state of a task, is also embedded in the concept definition and automatically applied.

In fewer words, the semantic construct is more understandable as it clearly relates the information it contains to the problem domain and is knowledge intensive as it embedds a variety of knowledge about the problem domain and the programming process.

More than that, the structured-object representation is easier to modify. Several kinds of semantic manipulations which ae useful for maintainance can be trivially accomplished in this representation in a fully automatic manner:

(1) Identifying and modifying the existing task states

(?) Identifying the processing associated to a given task state

(3) Identifying and modifying the error signaling part

(4) Identifying and modifying the state access mechanism.

On the initial LISP representation neither of these manipulations is easy to automate. The major reason is that in the semantic construct the relevant information is semantically tagged and thus easily accessible and interpretable. In the usual (LISP) representation only complex analysis of the code would eventually reveal the semantic role of the syntactic constructs used (e.g. the s variable, the cond or the prog). Another reason is that the semantic construct has a locality property [21]. Information spreading optimizations typical at the code level (such as the caching of the task state) have no place at this level.

B. Structured object support for semantic constructs

Three kinds of services are provided for programming with semantic constructs.

1) Organization into inheritance lattices. This allows semantic concepts to be efficiently composed by combining various features available from other constructs in the lattice.

2) Compilation. Semantic constructs can be efficiently compiled into executable code by attaching code generation methods activated by the instantiation processor. Code initially scattered on leaf instances of the instance tree is latter assembled in parent instances until the final code is produced.

3) Semantic editing. This is the profess by which the behaviour of a program is changed according to a given purpose. As semantic constructs explicitly represent semantic information, semantic editors can efficiently retrieve and interpret this information to achieve thei r goals.

Co n s) de r t he Ta s k Eligible .' routine from fig.1 which determines task eligibTlity for execution by the instantiation processor. Suppose one needs to produce a new version which would not consider not-yet as a possible answer to start-if and resume-if messages. (These may be needed for customizing the system to a specific application). First, this request can be reformulated as "remove the not-yet entry from the msg-pos-answ slot". To carry it out, a generic semantic editor for this type of semantic editing would have to perform (by itself) the following simple actions: (1) remove the specified entry from the msg-pos-answ slot; (2) retrieve, for each task state, the places where the removed value is used. As the use of message passing is restricted to a few semantic constructs ' such as Messaged singcase) defined in advance, this is quite easy to do, (3) edit the above found constructs. For MessagePassingCase-s (and similar ones) this amonts to removing the not-yet slot and eventually creating some new units, which take primitive operations in the language. Domain specific demons can augment this general editor by checking whether the suspended state of the task, which was set in this case. does not become superfluous. If it does, new editing is triggered this time on 1 ho Task abstract data type.

```
CTaskFligible?
 self (a SemanticProcedureMetaUnit
         supers (TaskProcess ingProc
                 MessagePass ingProtIn terpProc ) )
 msg (select-if resume-if)
 msg-pos-answ ((select-if t drop not-yet fail)
               (resume-if t drop not-yet fail))
 def au 1 t-msg-answ ((select-if t) ( resume--i f t))
 not-in-range-event (select-val-not-in-range )
 new (a MessagePasslngCase
        selector select-if
        send-to Task:Instance
        t <select task >
        drop <do not select)
        not-yet <make task suspended)
        fail <exit instantiation system)) ...J
```

Fig. 7: Task Eligible? unit

C. Discussion

When designing semantic constructs, the programmer must have already acquired a good understanding of the application domain. Semantic editors appear as local mutation mechanisms able to turn an existing construct into a class of different useful and meaningful constructs. What constitutes a meaningful mutation must be decided in advance for each construct. Meaningful mutations can be seen as modifying some previous assumption made during program design. This idea may be fruitful for further developments of this approach.

This explains why with semantic constructs the programmer is not designing a single program, but a class of potential programs any of which can become real by appropriate semantic editing. This helps enforce a kind of design for change.

The Programmmer s Apprentice [22] uses plans to capture middle level programming knowledge. Our semantic constructs *are* aimed more at representing problem specific knowledge. This makes semantic editing possible and distinct from what is ment in [233 by knowledge-based editing. This distinction also explains the use of a special plan language in [22] and the use of a more general representation language for semantic constructs. The Draco [24] system also uses problem oriented languages, but its major goal is to achieve reuasbiiity and the technology used qui te different.

D. The self generation layer

The approach presented in this section has been applied to several of the XRL components. At the instantiation level, the whole instantiation processor has been generated in this manner. The rule system employed by DBP was also constructed in this way.

Existing semantic editors can be used to modify the instantiation protocol , the instantiation system mechanism, for building interpreters for new types of operationalization rules and for modifying the existing ones.

The semantic editing idea has been further extended by associating an application analysis program which scans an XRL application knowledge base to see whether a simpler and more efficient XRL interpreter can handle it. This may happen e.g. when only a few of the capabilities provided by the instantiation protocol are actually used. If this is possible (decided by heuristic rules invocked over collected data), semantic editing is automatically applied to actually create the simplified XRL interpreter.

VI. FINAL REMARKS

While there exist several environments which exploit a structured object substrate for developing more declarative implementations of programming tools [2-6], XRL is different in the nature of its components. These components attempt to support higher level knowledge programming issues such as instantiation, descriptive programming and semantics-oriented support for procedural programming.

XRL has been used in several expert systems, mainly in the design area. Previous work investigated its applicability as an environment for design problem-solving [8,19]. The approach to procedural programming has been generalized to a semantics-oriented software development model supporting evolution and reusability [25].

Much work is still needed for better assessing the potential of description based programming, for developing more (kinds of) operationalization rules and more applications built on it. The integration of constructive and classificatory reasoning is also an important topic for future research.

More work is also needed for evaluating the measure in which procedural programming can be replaced with our object oriented tool and in general for assessing the possibilities of using semantic constructs in programming.

These developments are related to a longer range effort of creating a general software and knowledge engineering environment integrating declarative programming styles and tools of the kind discussed here.

REFERENCES

1 Doyle, J., Expert systems and the "myth" of symbolic reasoning. IEEE Trans.Soft.Eng. SE-11, No 11, Nov. 1985

2 Wright, J.M., Fox, M.S., Adam, D., SRL/1.5 Users Manual. T. R. CMU-RI, 1984

3 Bobrow, D.G., Stefik, M., The LOOPS Manual. T.R. KB-VLSI-81-13, Xerox Palo Alto , 1981

4 Lafue, G.M.E., Smith, R.G., A modular tool kit for knowledge management. IJCAI'85, Los Angeles, Vol I, pp. 46-52, 1985

5 Novak, G.S., GLISP, a LISP-based programming system with data abstraction. AI Magazine, Fall 1983, pp 37-47

6 Cannon, H.L., Flavors: a non hierarchical approach to object-oriented programming. AI lab. MIT, 1981

7 Barbuceanu, M., Trausan-Matu S., Molnar B., Integrating declarative knowledge programming styles and tools for building expert systems. T.R. KFKI-1987-02/M, Budapest, Hungary

8 Barbuceanu,M. An object-centered framework for expert systems in CAD. In J.S. Gero (ed) Knowledge engineering in computer-aided design, North-Holland 1985

9 Brachman, R.J., Levesque, H.J., The tractability of subsumption in frame-based description languages. AAAI'84, Austin Texas, pp 34-37

10 Brachman, R.J., Schmolze, J.G., An overview of the KL ONE knowledge representation system. Cognitive Science 9(2), 1985, pp 171-216

11 Brachman, R.J., Gilbert, V.P., Levesque, H.J., An essential hybrid reasoning system: knowledge and symbol level accounts of Krypton. IJCAI'85, Los Angeles, pp 532-539

12 Attardi,G.,Simi,M. A description oriented logic for building knowledge bases. TR ESP/86/3 1986

13 Rychener,M., PSRL: a SRL-based production rule system. Carnegie-Mellon Univ., dec. 1984

14 Mostow,J., Toward better models of the design process..AI Magazine, spring 1985, pp 47-57

15 Brown, D.C., Chandrasekaran, B., Expert systems for a class of mechanical design activity. In J.S. Gero (ed.) Knowledge engineering in computer-aided design, North-Holland 1985.

16 Maher, M.L., Fenves, S.J., HI-RISE: an expert system for the preliminary structural design of high rise buildings. In J.S. Gero (ed.): Knowledge engineering in computer-aided design. North-Holland, 1985

17 Kant, E., Barstow, D., The refinement paradigm: the interaction of coding and efficiency knowledge in program syntyesis. IEEE Trans. Soft.Eng. SE 7, 5, 1981

18 Clancey, W., Heuristic Classification. Artificial Intelligence, 27 pp 289-350, 1985

19 Barbuceanu, M., A domain-independent knowledge engineering architecture for CAD. 5-th Intl. Workshop on Expert Systems and Applications, Avignon, France 1985

20 Burton, R.R., Semantic grammar: an engineering technique for constructing natural language understanding systems. BBN Rep. No 3453, Cambridge Mass. 1976

21 Balzer, R. A 15 year perspective on automatic programming. IEEE Trans. Soft. Eng. SE-11, No 11, nov. 1985

22 Rich,C.,Shrobe,H.E.,Waters R.C. An Overview of the Programmer's Apprentice. IJCAI'79, Tokyo, 1979

23 Waters, R.C., The Programmer's Apprentice: Knowledge based program editing. IEEE Trans. Soft. Eng. SE-8, 1, 1982

24 Neighbors, J.M., The Draco approach to constructing software from reusable components. IEEE Trans. Soft. Eng., SE-10, No 5, sept 1984

25 Barbuceanu, M., Knowledge-based development of evolutionary and reusable programs. 6-th Intl. Workshop on Expert Systems and Applications, Avignon, France 1986