A Modular Tool Kit For Knowledge Management

GillesM E Lafue *Reid G. Smith*

Schlumberger Doll Research
Old Quarry Road
Ridgefield, CT 06877-4108
USA

ABSTRACT

We describe an integrated programming environment for developing knowledge-based systems. The environment contains a variety of general-purpose tools: a rule interpretation system, a semantic integrity manager, a task representation system and a file manager. Although the tools have different origins *(e g .* rule based systems, database management, process control), an object-oriented foundation lends modularity and consistency to the tool kit.

I. INTRODUCTION

By now the value of powerful tool kits for developing knowledge based systems is well-understood. Such tools act as *substrates* - computational bases that allow system builders to concentrate on the problems of acquiring and formalizing domain knowledge. In this paper we discuss a number of specific tools that we have found to be useful. We also argue for the utility of an object-oriented foundation that ties the individual tools together in a coherent fashion

The foundation for our tool kit is an object-oriented knowledge representation system called *Strobe* [5. 6]. It has been used in a variety of applications, most recently in *Crystal [9],* a system that supports interactive manual and automatic interpretation of well logs, (i e , measurements made in boreholes) This system was developed for two main classes of users, end users (*/ e* log interpreters) doing interactive well log interpretation, and interpretation program developers. It combines graphic display and manipulation of logs with graphic editors and menus It controls distributed execution of interpretation programs either on the Xerox workstations or on Ethernet connected Vaxes These programs have been written from a variety of computational perspectives (e.g.. statistical, pattern recognition, symbolic) and in a variety of programming languages. Strobe and Crystal provide the *glue* that binds these heterogeneous components together.

The Crystal knowledge bases incorporate a number of subsystems that are neither specific to the Crystal application nor to well-log interpretation. These subsystems are the tools of a modular, general-purpose knowledge-management tool kit that can be used to extend Strobe. Each tool is incorporated into a basic Strobe knowledge base by adding slots to one of the initial domain-independent objects, or by adding one or a few objects. Each tool is modular in that it doesn't require modifying slots or functions defined by other tools.

This paper describes a rule interpretation system, a semantic integrity manager, a task representation system, and a file manager for storing and retrieving objects These tools implement the interpretation of the declarative representations for the entities they manage rules integrity constraints, tasks, files The salient features of Strobe needed for the presentation of the tools are introduced in the next section Each tool is then described first through examples of the declarations it supports - drawn primarily from Crystal - and then by showing the objects that implement it

II. STROBE

A Strobe *knowledge base* is a collection of *objects* of two types *classes* which can be specialized, and *individuals* which cannot Objects are organized into taxonomic hierarchies (which may be tangled) along which properties are inherited (alternative inheritance paths are also supported)

A Strobe object has properties called *slots,* and a slot has properties called *facets* Both slots and facets may be inherited Some facets namely. *Value* and *Datatype,* are system-defined and exist for all slots, others are user defined Message-passing and event-oriented computation are supported.

Strobe supports multiple knowledge bases in the same address space Also, messages can be sent to objects in any knowledge base, even if that knowledge base is on another machine connected by Ethernet. (Indeed, any Strobe operation can be performed across the network.) To further support distribution and computational heterogeneity, Strobe had been implemented in a variety of languages' Interlisp-D on Xerox workstations. Mainsail. CommonLisp and C on Vaxes. This paper concentrates on the Interlisp-D version.

Interlisp-D Strobe has a display-oriented editor called *Impulse* [4]. This editor is itself built as a collection of Strobe objects. As a result, it can be specialized to suit the needs of Strobe applications, based on the types of declarations supported by the tools discussed here. An example of such a specialization is presented in [9].

A. User-defined Datatypes

A datatype in Strobe is itself implemented as an object. Datatyping provides another form of inheritance in that a slot may inherit some of its facets, typically operations, from its datatype. More precisely, receivers for messages sent to facets may be inherited from slots in the datatype object. Such slots are characterized by the fact that their name starts with the atom "DATUM ". For in stance, a message sent to the *Put* facet of a slot is forwarded to the *DATUM-Put* slot of the object implementing the datatype of the slot if no receiver can be found through standard taxonomic inhentance.

B. User-defined Facets

User-defined facets are useful for metadata encoding. They have been used extensively in Crystal to support communication between knowledge bases By agreeing on the meaning of a relatively small number of facets, knowledge bases can interpret each other's objects and slots without detailed knowledge of each other's domain of application (e g . object and slot names). For example, a part of an entity may be identified based on a *Role* facet set to *Part* rather than on the name of the part Each of our tools defines some facets and incorporates an interpreter for them

C. Initial Objects

A Strobe knowledge base starts with a few general objects which are organized in the generalization hierarchy shown in Figure II 1. DATATYPE is the ancestor of all datatype objects. Initially, it comes with a few slots that contain functions for implementing basic operations such as putting a value into a slot, getting, printing or editing a slot value The message handlers for some of these slots (eg, DATUM-Edit or DAIUM Pr int.) are defined in the descendants of DAIATYPL: BITMAP. TEXT, LISP (for Interlisp D functions and lambda expressions). EXPR (for arbitrary s expressions) and OBJECT (for slots whose value points to Strobe objects)

```
Object DATATYPE
Type Class
Generalizations. ROOT
  DATUM Edit
  DATUM Print.
  DATUM Get  sys/mgetvalue
  DATUM Put  sys/mputvalue
```

Figure II-1: Initial Strobe Knowledge Base Objects[1]

III. RULE INTERPRETATION

The rule interpreter [7] included in our tool kit applies rules in a forward-chained manner. The rule syntax supports a number of types of match variable instantiation in the left-hand side In addition, the system provides support for user extensions to the syntax. A form of rule compilation can be used to generate code and speed up rule application. Rules can be grouped into *Ruiesets* and control structures defined for attempting and firing rules associated with ruiesets. Uncertainty is currently handled according to the EMYCIN model [10]. There is also a simple mechanism for generating natural language rule translations Finally, and most important, rules written for this system can access domain knowledge encoded as Strobe objects.[2]

A. Declarations For Rule Interpretation

Each rule in the system is itself a Strobe object - an instance of the class *Rule* Figure III 1 shows a sample rule (taken in concept from the *Dipmeter Advisor* system [8])

```
Object: NormalFault9
Type: Individual
Generalizations Rule
  IF (Condition1 Condition2)
  THEN· (Action1 Action2)
  Condition1· (THERE EXISTS  Y NormalFaultZone)
  Condition2 (THERE EXISTS  Z RedPattern
               ($  ($LENGTH  Z) RedLength))
               ($ABOVE  Z  Y OverlapGap)
               ($PERPENDICULAR (THE Azimuth  Z)
                               (THE Strike  Y)
                               Tolerance))
  Action1  ($SPECIALIZE  Y to be a LateFaultZone)
  Action2  ($ASSIGN DirectionToDownthrownBlock of  Y to be
               (THE Azimuth  Z))
  TRANSLATION·
  IF
  (1) there exists an instance of the class
          NormalFaultZone ( Y). and
  (2) there exists an instance of the class RedPattern ( Z)
        such that the length of  Z · RedLength  and
        such that  Z is above  Y within |Overlap.Gap|. and
        such that the Azimuth of  Z is perpendicular to
          the Strike of  Y within Tolerance degrees
  THEN
  (1) specialize  Y to be a LateFaultZone
  (2) the DirectionToDownthrownBlock of  Y _the Azimuth of  Z
```

Figure 111-1: Late Fault Rule

Clauses may refer to Strobe objects (e.g., via the THE function, in Figure 111-1, which accesses a slot of an object) as well as other Lisp data structures. They may also bind variables that can be accessed during the matching and execution of the rule. We have defined an initial rule language of approximately 50 left-hand-side predicates and 10 right-hand-side actions. The rule language may be augmented as desired by end-users by defining Interlisp functions (and translation templates, if desired)

To match Norma IFau H9 against the database, the rule interpreter attempts to find a normal fault zone and a zone characterized by a signal pattern known as a red pattern, such that the two zones together satisfy the predicates associated with Condition2. The rule interpreter attempts to match the rule by sequentially instantiating the match variables. As each match variable is instantiated, the truth value of the clause is tested. If true. then the instantiation process continues, but if false, then the process backtracks to the last variable instantiated that has untried values in its domain and reinstantiates it to its next value.[3]

Figures showing objects are to be read as follows General information about the obiect (such as its synonyms, whether it is a class or an individual, its immediate generalizations) is shown first. Slots are then shown indented with respect to the object information, and when shown, the facets of a slot are indented with respect to the slot name. The value (if any) is printed following the slot name. Names in curly brackets are synonyms for the slot with which they are associated.

[2] The ability to integrate rules and structured objects as well as the ability to extend the rule syntax in a flexible way are not easily managed in existing rule interpreters, like OPS5[1] We have opted for the *flexibility* end of the flexibility/speed spectrum

Mark of Schlumberger

[3] Note that there may be several zone pairs for which the conditions are satisfied. To find them all, some form of iteration is required This is performed automatically, as desired, by the rule interpreter.

There are several forms of *quantified-condition* clauses. In the standard case, the domain of a match variable is the set of instances of a specified class (as in Figure III-1). Alternatively, it can be the set of instances of a class and of all its subclasses Other possibilities can be defined by the user Another type of quantified-condition clause that we have found useful is illustrated by the rule in Figure III-2 This rule expresses the fact that a tidal flat is characterized by a number of signal patterns known as blue patterns, that have alternating azimuth. We do not know *a priori* how many such patterns there will be - their number depends on the thickness of the flat. Set quantification in a single rule appears to provide a useful match to the way that variable thickness units are conceptualized and detected by our domain specialists

Rules are grouped into *Rulesets* A ruleset defines the control structure to attempt and fire the rules it contains It can eliminate the need for clauses often found in flat rule systems that are aimed at setting the context for rule application. Figure III 3 shows a ruleset used for stratigraphic interpretation of dipmeter data

The simplest ruleset contains a single list of rules called NORMAL-RULES. These rules are attempted one at a time according to their order in the list. Each time a rule fires, the content of the CONTROL-STRATEGY slot determines whether scanning is to continue with the next rule (ContinueAfterFiring) or is to begin again with the first rule (ReStartAfterFiring)[4] This iteration continues until no rules can be successfully fired (This default termination condition can be overridden in particular rulesets.)

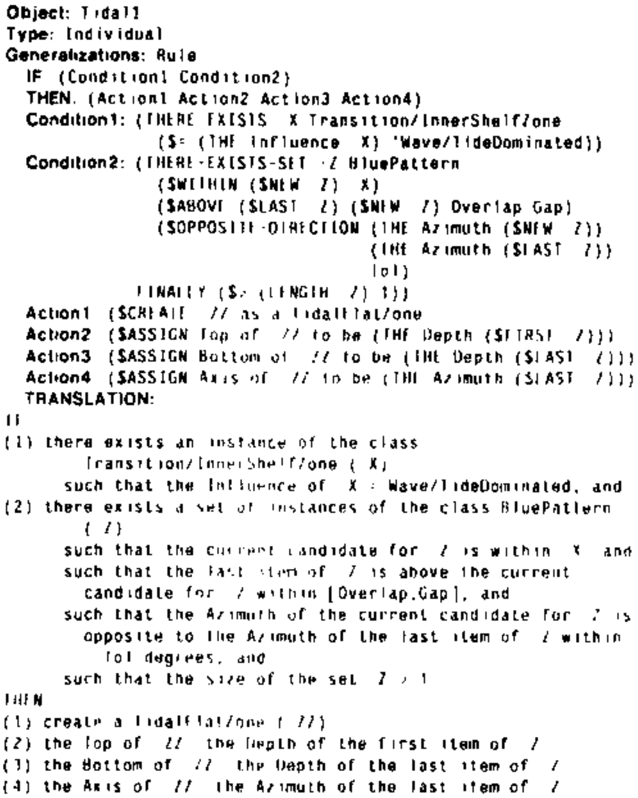
A ruleset can also contain the following lists of rules FIRE-ONCE-RULES can only be fired once for each invocation of the ruleset. They can be used to initialize the execution of a ruleset. FIRE-ALWAYS-RULES are attempted on each iteration regardless of the control strategy. They can be used to *respond* to the firing of other rules (e *g* . to update dependent relationships). Unlike NORMAL-RULES and FIRE-ONCE-RULES, FIRE-ALWAYS-RULES can be fired more than once with the same match variable bindings within a single ruleset invocation.

To assist in explanation and debugging, the execution history of each ruleset invocation is recorded as an instance of the ruleset class. It retains information such as the rules that were attempted and fired (together with their match variable bindings and the objects created or modified as a result of their firing).

B. Rule Interpretation Subsystem

The rule interpreter is implemented via the Rule object and the Ruleset object (Figure III 4) (Not all of their slots are shown)

An additional feature has proved quite useful for hypothesis testing. In applying a ruleset (or a rule), the caller is allowed to pass in a set of bindings. These bindings are analogous to lambda bindings and allow ruleset invocation to be considered as a form of function invocation. Match variable bindings may be included in this list, in which case, the rule interpreter tries to fill in the remaining instantiations.



```
Object: Tidal1
Type: Individual
Generalizations: Rule
  IF (Condition1 Condition2)
  THEN. (Action1 Action2 Action3 Action4)
  Condition1: (THERE EXISTS  X Transition/InnerShelfZone
                ($= (THE Influence  X) 'Wave/TideDominated))
  Condition2: (THERE-EXISTS-SET  Z BluePattern
                ($WITHIN ($NEW  Z)  X)
                ($ABOVE ($LAST  Z) ($NEW  Z) Overlap Gap)
                ($OPPOSITE-DIRECTION (THE Azimuth ($NEW  Z))
                                     (THE Azimuth ($LAST  Z))
                                     Tol)
              FINALLY ($> (LENGTH  Z) 1))
  Action1  ($CREATE  ZZ as a TidalFlatZone
  Action2  ($ASSIGN Top of  ZZ to be (THE Depth ($FIRST  Z)))
  Action3  ($ASSIGN Bottom of  ZZ to be (THE Depth ($LAST  Z)))
  Action4  ($ASSIGN Axis of  ZZ to be (THE Azimuth ($LAST  Z)))
  TRANSLATION:
IF
(1) there exists an instance of the class
        Transition/InnerShelfZone ( X)
      such that the Influence of  X = Wave/TideDominated, and
(2) there exists a set of instances of the class BluePattern
        ( Z)
      such that the current candidate for  Z is within  X  and
      such that the last item of  Z is above the current
        candidate for  Z within [Overlap.Gap], and
      such that the Azimuth of the current candidate for  Z is
        opposite to the Azimuth of the last item of  Z within
          Tol degrees, and
      such that the size of the set  Z > 1
THEN
(1) create a TidalFlatZone ( ZZ)
(2) the Top of  ZZ  the Depth of the first item of  Z
(3) the Bottom of  ZZ  the Depth of the last item of  Z
(4) the Axis of  ZZ  the Azimuth of the last item of  Z
```

```
Object: DEEP-MARINE-RULESET
Type:  INDIVIDUAL
Generalizations- Ruleset
   NORMAL-RULES: MARINE-20 MARINE-21 MARINE-22 MARINE-23
                 MARINE 24 MARINE-27
   CONTROL-STRATEGY- ReStartAfterFiring
```

Figure III-3: Deep Marine Ruleset

```
Object: Rule
Type:CLASS
Generalizations ROOT
   IF
   THEN.
   RULESET
   TRANSLATE: translate Rule
   TRANSLATION
   Apply: ApplyRule
   Match: MatchRule
   MatchAll. MatchRuleAll
   Execute. ExecuteRule
```

```
Object. Ruleset
Type-CLASS
Generalizations: ROOT
   NORMAL RULES:
   FIRE-ONCE RULES.
   FIRE-ALWAYS-RULES:
   CONTROL STRATEGY:
   TERMINATION-CONDITION
   KNOWLEDGE BASE:
   Apply  ApplyRuleset
```

Figure III-4: Rule Interpretation Objects

IV. SEMANTIC INTEGRITY MANAGEMENT

The integrity management system allows the user to define constraints on slots of objects, and to define actions to be taken in case of constraint violation or satisfaction (with appropriate defaults). The system analyzes the constraints and derives the information it needs to check them at run time *(i.e.,* the constraint variables and the operations that require checking). Currently, our constraint language is a combination of Interlisp-D and Strobe.

---

[4] The ReStartAfterFiring option is useful when the actions of an individual rule could interact with the conditions of other rules It is often used in combination with a rule ordering that places the most specific rules before the most general rules. The ContinueAfterFiring option is useful when individual rule actions are independent or additive (It is the default strategy.)

A. Declarations For Integrity Management

We distinguish several types of integrity constraints. A *single-slot constraint* involves a single-slot. A *datatype constraint* applies to all slots having a particular datatype. A *multi-slot constraint* involves several slots, in one or several objects. Since slots can be set-valued, constraints are divided into *element constraints* (that apply to the elements of a slot value) and *set constraints* (that apply to a slot value as a set).

Two alternatives for encoding a constraint are supported. *0)* as slots of an obiect and *(n)* as facets of a slot In this section, we discuss constraints encoded as objects For a presentation of slot encoding, and criteria for choosing between the two alternatives, see [2] and [3)

Figure IV 1 shows the object that defines a well location in terms of a town, county, state, country and continent. The Continent slot is subject to a (single-slot) constraint that its value must belong to the set of names enumerated in the Candidates facet The constraint is implemented in the ContinentConstraint object (Figure IV 2) whose *Condition* slot encodes the constraint definition and *Correction* slot the correction of violations (here, simply an error message)

```
Object. Weillocation
Type Class
Generalizations OBJECT
  Well
  TOWN {Township}
  COUN {County Parish}
  STAT (State Province}
  NATI {Nation Country}
  CONT {Continent}
    Datatype. IXPR
    Candidates- (Europe North-America South-America Asia Africa
                 Australia)
    PutElementConditions. ContinentConstraint
    AddElementConditions: ContinentConstraint
```

Figure IV-1: The WellLocation Obiect

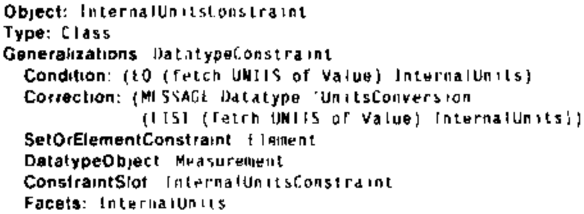
```
Object ContinentConstraint
Type: Class
Generalizations: SingleSlolConslraint
  Condition. (MEMBER Value Candidates)
  Correction: (Error Value "is not one of " Candidates)
  SetOrElementConstraint. Element
  ConstrainedObject. Well location
  ConstrainedSlot: Continent
  Facets: Candidates
```

Figure IV-2: Single-Slot Constraint On Continent

Figure IV-3 shows the Measurement datatype object and one of its constraints: the InternalUnitsConstraint, implemented in the object shown in Figure IV-4. The constraint states that if slots whose datatype is a measurement specify the units for their values-their internal units-the values they are assigned must be in those units. The *Correction* slot contains the code to make the conversion if necessary.

```
Object: Measurement
Synonyms: DimensionedQuantity
Type: Class
Generalizations: DATATYPE
  UnitsConversion:
  InternalUnitsConstraint: InternalUnitsConstraint
    Datatype: DatatypeConstraint
  PutElementConditions. InternalUnitsConstraint
  AddElementConditions: InternalUnitsConstraint
```

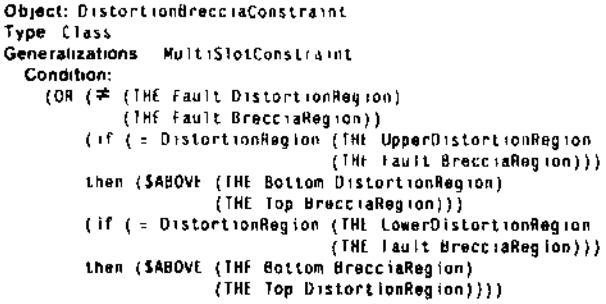
Figure IV-3: The Measurement Object



```
Object: InternalUnitsConstraint
Type: Class
Generalizations DatatypeConstraint
  Condition: (EQ (fetch UNITS of Value) InternalUnits)
  Correction: (MESSAGE Datatype 'UnitsConversion
                 (LIST (fetch UNITS of Value) InternalUnits))
  SetOrElementConstraint Element
  DatatypeObject Measurement
  ConstraintSlot InternalUnitsConstraint
  Facets: InternalUnits
```

Figure IV-4: Datatype Constraint For Internal Units

Figure IV-5 shows the *Condition* slot of an object that implements a multi-slot constraint involving several objects The constraint is between different regions, or components, of a geological fault, referred to as the upper and lower distortion regions (or blocks) and the breccia region (the zone between the blocks characterized by crushed rocks) It states that the upper distortion region of a fault is above its breccia region which, in turn, is above its lower distortion region. It assumes that there exist (i*)* Fault objects with UpperDistortionRegion, LowerDistortion-Region and BrecciaRegion slots, and *(n)* DistortionRegion and BrecciaRegion objects with Fault, Top and Bottom slots.



```
Object: DistortionBrecciaConstraint
Type Class
Generalizations MultiSlotConstraint
  Condition:
    (OR (≠ (THE Fault DistortionRegion)
           (THE Fault BrecciaRegion))
        (if (= DistortionRegion (THE UpperDistortionRegion
                                     (THE Fault BrecciaRegion)))
        then ($ABOVE (THE Bottom DistortionRegion)
                     (THE Top BrecciaRegion)))
        (if (= DistortionRegion (THE LowerDistortionRegion
                                     (THE Fault BrecciaRegion)))
        then ($ABOVE (THE Bottom BrecciaRegion)
                     (THE Top DistortionRegion))))
```

Figure IV-5: Multi-Slot Constraint Between Fault Regions

The user fills the *Condition* slot and optionally, the *Correction* and *Action* slots (the latter states a side effect of constraint satisfaction). These slots can contain function names, lambda expressions or s-expressions. They can reference slots and facets as free variables rather than by using Strobe access functions. In single-slot and datatype constraints, *Value* refers to the current slot value. In multi-slot constraints involving several objects, the *THE* function references slots in relation to their objects.

The checking declarations generated and used by the system basically consist of identifiers of the constraint variables to allow *d)* insertion of triggers from the variables to the constraints at analysis time and *(n)* efficient binding of the variables at run time.

A single-slot constraint object has a *ConstramedQbject* slot and a *ConstrainedSlot* slot. A datatype constraint object has a *DatatypeObject* slot and a *ConstraintSlot* slot. Since these constraints can involve facets other than *Value (e.g..* InternalUnits in InternalUnitsConstraint) such facets are declared in a *Facets* slot. A *Slots* slot in a datatype constraint declares the slots of the datatype object involved in the constraint, and in a multi-slot constraint, it declares the constrained slots together with their respective objects.

A trigger associated with a slot is implemented by a facet that points to single or multi-slot constraints. The facet name indicates the operations on the slot that require checking *{Put, Add, Remove)*[5]. It also indicates whether the constraints are set or element constraints This allows the user to reset the order in which constraints are checked (e *g* element constraints before set constraints) and the system to efficiently order the constraints to check at run time For example, the *PutElementConditions* facet of the Continent slot (Figure IV-1) indicates that ContinentConstraint must be checked when a value is put in the slot.

A trigger associated with a datatype object is a slot that points to the constraints for the datatype. Its name encodes the same information as trigger facets The *AddElementConditions* slot of Measurement shows that InternalUnitsConstraint must be checked when a value is added to a slot whose datatype is some measurement.

B. Integrity Management Subsystem

The integrity management system consists of slots added to the *DATATYPE* object (Figure IV-6) and of constraint objects. The constraint objects are organized in a taxonomic hierarchy. The *Constraint* object is shown in Figure IV-7.

The *DATUMPut, DATUM Add.* and *DATUM-Remove* slots of *DATATYPE* contain the operations for which integrity may be checked. These operations are also declared in the *OperationsWithIntegrity* slot of *Constraint DefaultOperations* declares the default operations for which integrity is checked

```
Object: DATATYPE
Type: Class
Generalizations: ROOT
  DATUM-Edit:
  DATUM-Print:
  DATUM-Get. sys/mgetvalue
  DATUM-Put: DatatypePut
  DATUM-Add: DatatypeAdd
  DATUM-Remove: DatatypeRemove
  DATUM-AnalyzeConstraints DatatypeAnalyzeConstraints
  ConstraintCheckingOrder (DatatypeElementConstraints
                           ElementConstraints
                     .     DatatypeSetConstraints
                           SetConstraints)
```

Figure IV-6: DATATYPE Object For Integrity Management

```
Object: Constraint
Type: Class
Generalizations: DATATYPE
Specializations: (SingleSlotConstraint DatatypeConstraint
                  MultiSlotConstraint)
  OperationsWithIntegrity (Put Add Remove)
  DefaultOperations (Put Add)
  Analyze:
  Verify:
```

Figure IV-7: The Constraint Object

The *DATUM-AnalyzeConstraints* slot of *DATATYPE* and the *Analyze* slot of *Constraint* provide alternative ways of analyzing constraints. The former is a message handler for analyzing constraints encoded in slots. The latter is a message handler for analyzing constraint objects. For example, an *AnalyzeConstraints* message sent to the PutElementConditions slot of Measurement results in filling the *DatatypeObject, ConstraintSlot* and *Facets* slots of **InternalUnitsConstraint**. An *Analyze* message to the InternalUnitsConstraint object results in filling the same slots as well as the *PutElementConditions* and *AddElementConditions* slots of Measurement.

The *Verify* slot in *Constraint* verifies whether a hypothetical value for a slot violates the constraints that apply to the slot. Instead of executing the corrections associated with the violated constraints, it returns an association list of the names of those constraints and the bindings of their variables. The *ConstraintCheckingOrder* slot declares the default order in which constraints are checked and can be reset in any datatype.

V. TASK REPRESENTATION

Declarative task representation has been successfully used to capture component function and structure in a number of domains[1] hardware design, fault detection, well-log interpretation Our motivation for task declaration is to provide a structure within which (1*)* a knowledge-based system can reason about tasks, *(n)* a unified mechanism can control task execution, and *(III* code written from a variety of computational perspectives and in a variety of programming languages can be integrated An example is described in [9]. It shows how the Crystal knowledge base responsible for the user interface interprets information about tasks to guide the user through their execution and how it prompts him for the necessary input To date, we have concentrated on task/subtask relationships, data description and control flow

A. Declarations For Task Representation

In our formalism, task declarations are made in subclasses of the *Module* object. The execution history of a task is recorded as an instance of its class (analogous to ruleset invocation) Figure V 1 shows some of the slots of a task called Eigen which represents a principal component analysis on the logs identified in the ActiveLogs slot for the well identified in Well from TopDepth to BottomDepth. Among the outputs are principal component logs, represented by the PCLogs slot

A task can be a substask of another task, which is called its *abstraction.* It points to that abstraction via a slot whose *Role* facet is set to *Abstraction* For example, the Faciolog slot of an Eigen instance points to an instance of a module, called faciolog" - a program that finds zones of similar log responses in a well, and whose first subtask is the principal component analysis carried out by Eigen Conversely, a task points to its subtasks via slots whose *Role* facet is set to *Expansion.* Slots representing expansions also have an *Order* facet that indicates the relative (partial) order in which each expansion is normally to be executed. (Eigen has no expansions.)

The slots representing input and output parameters of a task are denoted by a *Role* facet set to *Port* and a *Direction* facet set to *In, Out* or *(In Out).* These slots also have an *Order* facet that indicates to the system the relative (partial) order in which each input parameter should get its value. The *Origin* facet identifies where the value for the slot can be obtained. It may *(i)* identify the user (which in Crystal causes the user interface knowledge base to take charge); *(ii)* specify a slot of another object; or *(HI)* indicate

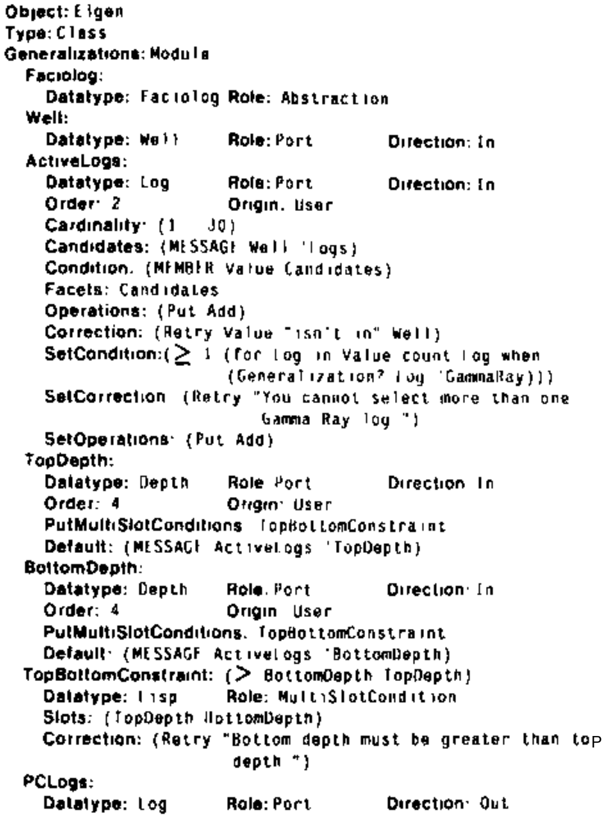
---

Currently, the identification of the operations that may cause a constraint violation is based on heuristic, rather than formal, analysis, and can be overridden by the user

Mark of Schlumberger

that the task will compute the value itself The *Default* facet contains an s-expression that evaluates to a default value, and the *Candidates* facet evaluates to a set of possible values. These two facets are used by the user interface knowledge base in its prompting of the user (e.g., by presenting a menu if there are candidate values). Other facets are discussed in [9].

The other facets associated with an input port are for integrity management. The ActiveLogs slot, for example, is subject to an element constraint that each value be drawn from the logs associated with the Well, and to a set constraint such that no more than one log can be of type GammaRay. Furthermore, its Cardinality is limited to 30.

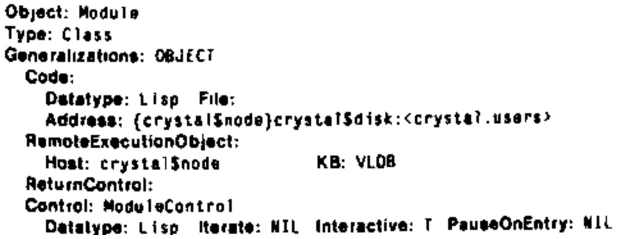


```
Object: Eigen
Type: Class
Generalizations: Module
  Faciolog:
    Datatype: Faciolog Role: Abstraction
  Well:
    Datatype: Well      Role: Port          Direction: In
  ActiveLogs:
    Datatype: Log       Role: Port          Direction: In
    Order: 2            Origin. User
    Cardinality: (1   30)
    Candidates: (MESSAGE Well 'logs)
    Condition. (MEMBER Value Candidates)
    Facets: Candidates
    Operations: (Put Add)
    Correction: (Retry Value "isn't in" Well)
    SetCondition:(≥ 1 (for Log in Value count log when
                      (Generalization? log 'GammaRay)))
    SetCorrection (Retry "You cannot select more than one
                         Gamma Ray log ")
    SetOperations: (Put Add)
  TopDepth:
    Datatype: Depth     Role Port           Direction In
    Order: 4            Origin: User
    PutMultiSlotConditions TopBottomConstraint
    Default: (MESSAGE ActiveLogs 'TopDepth)
  BottomDepth:
    Datatype: Depth     Role. Port          Direction: In
    Order: 4            Origin User
    PutMultiSlotConditions. TopBottomConstraint
    Default: (MESSAGE ActiveLogs 'BottomDepth)
  TopBottomConstraint: (> BottomDepth TopDepth)
    Datatype: Lisp      Role: MultiSlotCondition
    Slots: (TopDepth BottomDepth)
    Correction: (Retry "Bottom depth must be greater than top
                      depth ")
  PCLogs:
    Datatype: Log       Role: Port          Direction: Out
```

Figure V-1: Eigen Module

B. Task Declaration Subsystem

The mechanism to control task execution is implemented in the Module object (Figure V-2). (Only some slots are shown.)



```
Object: Module
Type: Class
Generalizations: OBJECT
  Code:
    Datatype: Lisp  File:
    Address: {crystal$node}crystal$disk:<crystal.users>
  RemoteExecutionObject:
    Host: crystal$node             KB: VLOB
  ReturnControl:
  Control: ModuleControl
    Datatype: Lisp  Iterate: NIL  Interactive: T  PauseOnEntry: NIL
```

Figure V-2: Module Object

The computation carried out by a task may execute either on the Xerox workstation or on a remote machine, in a language other than Interlisp-D. In the former case, the computation is specified in the Code slot, which typically contains a function name. In the latter case, it is specified in the RemoteExecutionObject slot The value of that slot points to a Strobe object on the remote machine (identified in the *Host* facet) which is responsible for calling the foreign language program That object (written in Mainsail. CommonLisp, or C Strobe if it resides on a Vax) exchanges input and output parameters with the current module obiect written in Interlisp-D Strobe.

The ReturnControl slot identifies where control is to be passed next (in terms of a host, knowledge base, object, slot and facet)

A task executes when its Control slot receives a message. This slot contains the function that implements the task execution control mechanism. Basically, that function *(/)* acquires the input parameters; *(n)* instantiates and executes expansions as required, if there are expansions, or else executes the task computation, and *(m)* passes control to the next module. Dynamic alteration of control flow is supported by resetting ReturnControl slots on the fly. Note also that the Control slot has facets to modify control flow (e *g.,* to iterate through a task or to pause for interaction before starting a task) Of course, the default values for these facets defined in the Module object can be overridden in its specializations

VI. FILE MANAGEMENT

Strobe manages objects in virtual memory At the end of a session, all objects in a knowledge base are generally stored on the same file. Subsets of objects from a knowledge base may also be loaded and stored. The file management tool extends this basic capability in that (i*)* it allows more generality in specifying the subsets of objects (by description as well as by name), and *(n)* it keeps track of the files on which such collections are stored. Our goal is to provide DBMS like facilities to cope with increasing numbers of objects as knowledge bases scale up.

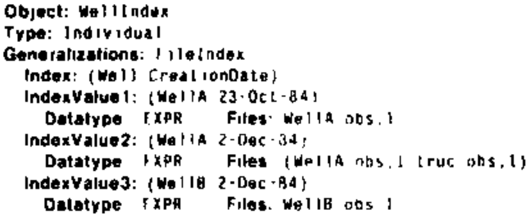
A. Declarations For File Management

The filing mechanism is implemented via *file indexes.* A file index is defined as a conjunction of slot names. It maps values of that conjunction into file names. For example, a file index may be defined by the conjunction (Well CreationDate) and an index value may be (WellA 23-Oct-84). The object implementing that file index associates (WellA 23-Oct-84) with the name of one or several files that contain objects whose Well slot value is WellA and whose CreationDate slot value is 23-Oct-84.

Figure VI-1 shows a file index whose conjunction, defined in the Index slot, is (Well CreationDate). The slots IndexValue1. IndexValue2 and IndexValue3 represent index entries, *i.e.,* tuples of the mapping between index values and file names. The *Value* facet of such a slot is an index value, *e.g.,* (WellA 23-Oct-84), and its *Files* facet points to the files that contain objects corresponding to its value[6]. Slots implementing index entries are created and managed automatically by the system and are of no more concern to the user than the implementation of

An index value may point to several files because it is our policy to avoid duplication of objects on several files As a result, an object corresponding to two index values (for two different indexes) is stored in only one of the two corresponding files, and that file must be pointed to by the other index value.

B trees in a DBMS. The user need only be concerned with loading and storing objects, not with the system's implementation of those operations

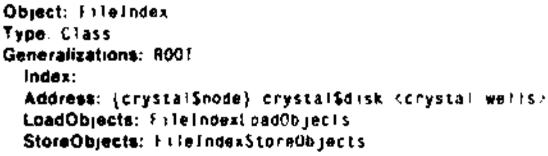


```
Object: WellIndex
Type: Individual
Generalizations: FileIndex
  Index: (Well CreationDate)
  IndexValue1: (WellA 23-Oct-84)
    Datatype  EXPR      Files: WellA obs.1
  IndexValue2: (WellA 2-Dec-84)
    Datatype  EXPR      Files (WellA obs.1 truc obs.1)
  IndexValue3: (WellB 2-Dec-84)
    Datatype  EXPR      Files. WellB obs 1
```

Figure VI-1: WellIndex object

B. File Management Subsystem

The file management subsystem is implemented in the FileIndex object (Figure VI 2) Its specializations are individual user defined objects representing file indexes such as Well Index Address specifies the host, device, and directory where the files are actually found. LoadObjects contains a function that takes an index value as argument and loads the objects corresponding to that index value Similarly, StoreObjects stores the objects corresponding to an index value.



```
Object: FileIndex
Type. Class
Generalizations: ROOT
  Index:
  Address: {crystal$node} crystal$disk <crystal wells>
  LoadObjects: FileIndexLoadObjects
  StoreObjects: FileIndexStoreObjects
```

Figure VI-2: FileIndex Object

VII. CONCLUSION

We have described the implementation of knowledge management tools for Strobe knowledge bases and presented examples of the capabilities they offer. Each tool is confined to a few general domain independent objects which can be added to an initial knowledge base. The addition of a new tool is modular in that it consists only of defining new objects or new slots of an existing object. Figure VII-1 shows the initial taxonomic hierarchy of a knowledge base incorporating all tools described in this paper

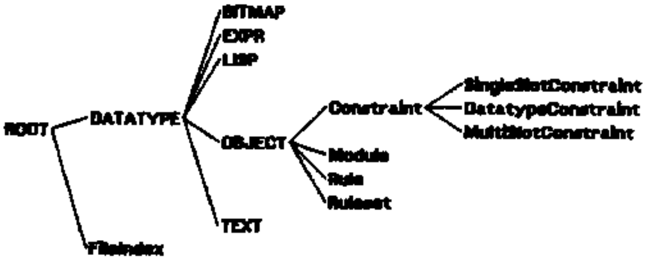


Figure VII-1: Initial Objects For Knowledge Management Tools

Implementation of the tools has been unified through an object-oriented foundation. This foundation also helps to unify access to the tools - through invocation via message. This is a simple, yet powerful concept that helps to integrate objects, rules, tasks, and procedures.

Tool kits such as ours offer a number of alternative styles of programming: Rulesets, modules, constraints, and procedures. While they do help integrate these various styles, criteria for selecting among the alternatives for any given task are not always clear. For instance, a computation to be carried out as the result of an operation on a slot could be encoded as a constraint (possibly with maintenance actions), or as a ruleset, or as a module whose invocation is triggered by a demon associated with that slot. Our intention, then, is to use the tool kit both as a development vehicle for knowledge based systems and as an exploration vehicle for seeking selection principles.

ACKNOWLEDGMENTS: We gratefully acknowledge the Crystal team for providing the context for this work. Eric Schoen built Impulse, parts of the task representation system and lower level tools such as the networking and Strobe Vax execution environ ment. Robert Young made a number of valuable suggestions.

REFERENCES

1. C. LForgy. *The OPS5 User's Manual* Tech. Rept. CMU-CS-81-135, Carnegie-Mellon University, July, 1981.

2. Lafue G. M. E *Integrity Constraints for Strobe Knowledge Bases.* Research Note SYS-85, Schlumberger-Doll Research, 1985

3. G. M. E. Lafue and R. G. Smith. Implementation of An Integrity Manager With A Knowledge Representation System. *Expert Database Systems,* 1985.

4. E Schoen and R. G. Smith Impulse, A Display-Oriented Editor for Strobe. *Proceedings of the National Conference on Artificial Intelligence,* August, 1983, pp. 356-358

5. R. G. Smith. Strobe. Support For Structured Object Knowledge Representation. *Proceedings of the Eighth internationalJoint Conference on Artificial Intelligence.* August, 1983, pp. 855 858.

6. R. G Smith. *Structured Object Programming In Strobe.* Research Note SYS-84-08, Schlumberger Doll Research, March. 1984

7. R G. Smith. *Programming With Rules In Strobe.* Research Note SYS-84-12. Schlumberger Doll Research, December, 1984

8. R. G. Smith and R. L. Young. The Design Of The Dipmeter Advisor System *Proceedings of the ACM Annual Conference,* ACM, New York, October, 1984, pp. 15-23.

9. R G. Smith, G. M. E. Lafue. E Schoen. and S. C Vestal. "Declarative Task Description As A User Interface Structuring Mechanism." *Computer 17,* 9 (September 1984), 29-38.

10. W. VanMelle, A. C. Scott, J. S. Bennett, and M Peairs. *The Emycin Manual.* Tech. Rept. STANCS-81 885(HPP-81-16), Dept. Of Computer Science. Stanford University, October, 1981.