The Workshop System - A Practical Knowledge-Based Software Environment

Geoffrey M. Clemm

Evolutionary Software Inc.
55 Commonwealth Road, Watertown, Mass. 02172

Abstract

The Workshop System is a programming environment designed to support teams of programmers working concurrently on large software projects. An essential feature of the Workshop System is the storage of all information from the software project as fine grained objects in a shared database. In order to allow effective usage of this potentially overwhelming amount of information, the Workshop System provides a rule-based language called SE-KRL for specifying both the software objects and the software process for a given domain. SE-KRL programs can then be written to automate mechanical aspects of the software development process as well as to guide the essential creative activity of a software engineer.

0

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1988 ACM 0-89791-290-X/88/0011/0055 \$1.50

1 Introduction

The term “practical” is much abused in the software environment community. In its most abused form, it simply means that the authors believe the system could be implemented. More commonly, it means that at least some aspect of the system has been implemented. For the purposes of this article, a more restrictive definition is intended : *A software environment is practical if it can be implemented using current technology and if the benefits to the user of the system outweigh the costs.* Note that the implementation must use current technology - otherwise at best one can only say that the system might become practical. Furthermore, note that the costs of a system are as important as the benefits - if a system requires PhD expertise and AI workstations, it will be impractical for most commercial software development. What distinguishes the Workshop System from other knowledge-based software environments is that it is designed to be practical for the development of commercial software.

In order to help satisfy the requirement that it be a *practical* knowledge-based software environment, the Workshop System is designed to interface with the Odin System[3] for all computation intensive operations. The Odin System is an extensible file-based ob-

ject manager, designed to maximize the time and space efficiency of an arbitrary tool suite. This allows the effective use of existing software tools to perform operations that are not or can not yet be efficiently implemented in either the knowledge-based system or the underlying commercial database.

In the current implementation, the Workshop System provides knowledge-based support for all front-end activities in the software environment, such as editing, configuration management, interactive testing, and most activities that fall under the description of *process programming*[5]. For back-end activities such as compiling, complex program analysis, and regression testing, the Workshop System automatically passes the appropriate objects to the Odin System, which then returns the results back to the Workshop System. As the efficiency of knowledge-based technology improves, functionality will be transferred from the Odin System into the Workshop System whenever practical.

2 Background

The conceptual predecessor of the Workshop System is the CLF system[2] developed at ISI. Knowledge in CLF is stored and applied by the AP5 database system[4], also developed at ISI. The CLF system distinguishes itself among knowledge-based environments by being a practical research environment - in particular, CLF has proven to be the environment best suited to developing future versions of CLF itself.

The main motivation for a new system was the desire to extend the practicality of knowledge-based software environments from the AI research world into the commercial world. Among the problems not directly addressed in the CLF system are problems of scale (such as millions of lines of code, large dispersed programming teams, and multiple

concurrent system versions and configurations) and problems of limitations in the commercial world (in areas such as programmer sophistication and computer resources).

Some of the characteristics of the Workshop System that deal with these problems and that are not found in the CLF system are :

- the use of a shared project database designed for concurrent access by large numbers of users (the CLF database is a single user AI-database stored in the virtual memory of the user's workstation).
- implementation designed for standard C-based workstations (CLF requires a lisp-based AI workstation for effective use).
- a software model designed to support the sophisticated version control required in large software development efforts.
- the use of knowledge representation language specifically designed for software engineering knowledge (CLF provides a general purpose rule-based language).

The Workshop System's software engineering knowledge representation language is perhaps the most interesting item in this list, and it will be the focus of this article.

3 SE-KRL : A Software Engineering Knowledge Representation Language

In the Workshop System, knowledge is specified in the language SE-KRL (pronounced *see-krull*), a knowledge representation language specifically designed for the specification of software engineering knowledge. In

SE-KRL, there are three basic categories of knowledge that can be captured - what kinds of objects can exist in the software environment, what operations can be applied to these objects, and what are the rules that govern the behavior and guide the development of software objects.

3.1 Workshop System Architecture

In order to understand SE-KRL, a rough understanding of the basic architecture of the Workshop System is required. The Workshop System is composed of four kinds of processes : Global Database processes that contain all the information about a given software project, Workshop processes that apply the Workshop System's software engineering knowledge, Editor processes that are domain specific user interfaces that support the creation and manipulation of software objects, and Studio processes that are domain specific processes for evaluating and testing the software objects (e.g. a debugger). For domains where both the manipulation and evaluation of the software objects can occur in the same process, the Editor and Studio processes will be identical.

Normally, each user will have a Workshop process and a collection of Editor and Studio processes on his local workstation. The Global Database process could reside on his workstation, but for large projects it is expected that a "database-server" machine would be reserved for Global Database processes. In the Workshop System, the only process communication is between a Workshop process and its local Editor and Studio processes, and between a Workshop process and a single¹ Global Database process - there

¹the Global Database process might access a distributed database, and therefore be implemented as a large number of cooperating processes, but this would not be visible in its interface to a Workshop process.

is no direct communication between Editor, Studio, and Global Database processes. This restriction ensures that the knowledge-based Workshop process has complete control of the information in the Workshop System.

3.2 Software Objects

The notion of object in the Workshop System is a very general one. In addition to the standard objects such as requirements, specifications, code, and testing information, things such as people, the processes that are used to create objects, and the software engineering knowledge itself are all instantiated in the Workshop System as software engineering objects.

Every object is classified as being an instance of a specific *class*, where the class itself is a software object. The classes are organized into an inheritance hierarchy, where a class by default inherits the characteristics of each of its superclasses.

An object is composed of a set of *attributes*, where an attribute consists of a *slot-value pair*. The *slot* of an attribute specifies the kind of information being stored in the *value* of the attribute. Each slot itself is an object (an instance of Slot-Class), where each slot object has slots such as *Domain* and *Range*, specifying what classes of objects can have this slot, and what classes of entities can be values of this slot, respectively. Following the inheritance hierarchy, if a given class is in the domain or range of a slot, all subclasses of that class are also in the domain or range, respectively, of that slot.

A new software object class is specified in SE-KRL with the DefClass defining form. In this form, some or all of the following class characteristics can be specified :

Superclasses the existing classes whose characteristics are to be inherited.

Parser if the object has a textual definition, the function that will parse a textual definition and return a syntax tree for the object.

Component-Slot if the object is an aggregate, the slot that contains the components of the aggregate.

Initial-Source if the object has a textual definition, a reasonable initial value for this definition.

Load-Priority if the object can be installed (i.e. compiled or loaded), how early the object should be installed (with respect to other classes of objects that can be installed).

Attributes boolean valued attributes of the class (such as whether or not a user can create new instances of this class).

Literals if the class is an enumerated type, the literals of the enumerated type.

An example of a SE-KRL class definition would be :

```
(DefClass Function-Class
:superclasses (Program-Object-Class
                Lisp-Class)
:initial-source "(DEFUN ^A ()~% NIL)"
:load-priority -10
:attributes User-Object)
```

This defines a class, Function-Class, that inherits the properties of both Program-Object-Class and Lisp-Class. In addition, the initial source text for a function, the load-priority, and the class attribute "User-Object" are specified.

A new software object slot is specified in SE-KRL with the DefSlot defining form. In this form, the Domain and Range classes of the slot are specified, as well as some or all of the following :

Inverse the inverse of the slot (e.g. the inverse of the Has-Element slot is the Element-of slot).

Default-Value the default value of the slot (if this is the symbol "query", the user is prompted for the value).

Attributes various binary valued attributes of the slot - e.g.

Required slot must exist;

Single-Value the slot must have at most one value;

Can-Change the user can change values of this slot;

Hidden the slot should not be displayed when the object is displayed.

Possibility-Generator function that generates a list of current objects that could reasonably be values of this slot.

An example of a SE-KRL slot definition would be :

```
(DefSlot Has-Documentation
:domain Global-Object-Class
:range String-Class
:attributes (Single-Value Can-Change))
```

This defines the slot *Has-Documentation* to be a single-valued, user-modifiable slot, mapping Global-Objects into Strings.

3.3 Software Object Operations

A new software object operation is specified in SE-KRL with the DefOperation defining form. In this form, the following operation characteristics can be specified :

Domain the class of objects to which this operation can be applied.

Object-Test additional properties a specific instance of the domain class must satisfy in order for the operation to be appropriate.

Application-Test decides whether operation can be applied to the object with the specified arguments.

Function-Body what to invoke if all tests succeed.

This explicit partitioning of knowledge about an operation (instead of simply providing a function), is critical for effective application of this knowledge. In particular, the Workshop System can use this information to tailor menus to be appropriate for a specific object, and to act appropriately when operations on arbitrary groups of objects are requested.

An example of a SE-KRL operation definition would be :

```
(DefOperation Do-Validate (?Workshop)
:mouse-line "Validate modifications"
:domain Workshop-Class
:test (AND
  (Is-Fact ?Workshop
    Has-Update-To ?Object)
  (Is-Fact ?Workshop
    Validation-Status Unknown))
:body (Perform-Validation))
```

This specifies that the *Do-Validate* operation applies to the Workshop process itself, but only if there are modifications and the validation status is unknown.

3.4 Software Object Rules

Although classes, slots, and operations would be sufficient to specify the software process, the software knowledge that guides the software process is often most conveniently specified in the form of rules. There are two aspects of process automation : the automa-

tion of mechanical processes and the prevention of detectable errors. All process automation in the Workshop System is performed by the Workshop process. Based on the architecture of the Workshop System, there are four categories of information flow that can be automated : within the Workshop process, between the Workshop process and the Global Database process, between the Workshop process and an external process (such as an editor or mail system), and between the Workshop process and the human user. The rule defining forms for these categories are named DefCoordination, DefDownloading, DefAutomation, and DefNotification respectively.

In the Workshop System, detectable errors consist of inconsistencies in the software objects being produced. A fifth rule defining form, DefAbort, is provided for specifying these inconsistencies. An additional more stringent level of error detection can be specified for the Global Database through the DefInvalidation defining form. This additional layer of error specification reflects the contrast between the Workshop process within which the support for creativity requires a significant degree of freedom, and the Global Database process within which project-wide comprehensibility requires a significant degree of consistency. In addition, these two levels of consistency corresponds to the fact that a software engineer must perform his work incrementally, which usually implies that only at the end of a sequence of operations can the software database return to a strictly consistent state.

In total, there are six kinds of rules that control and guide the flow of information in the Workshop System : DefDownloading rules that affect the flow from the Global Database to the Workshop process, DefCoordination and DefAbort rules that affect the flow within the Workshop process, and DefInvalidation, DefAutomation, and DefNotifica-

tion rules that affect the flow from the Workshop process to the Global Database, to external software systems, and to the human user, respectively.

In order to efficiently implement these rules, the Workshop System uses a commercial forward chaining production system. In the current version of the Workshop System, this production system is ART[1], a product of Inference Corporation. SE-KRL objects are translated into ART facts; SE-KRL rules are translated into ART rules; and the ART interpreter is invoked.

The advantage to a user of using SE-KRL, rather coding directly in ART, is that ART is a general purpose expert system language, most of whose features are not relevant (for functional and practical reasons) for the specification of software engineering knowledge. To effectively use ART, a user would need to be an expert in knowledge-engineering and software environment design, in addition to being an expert in software engineering. In SE-KRL, the knowledge-engineering and software environment knowledge has been designed into the SE-KRL language itself, requiring only software engineering knowledge on the part of the user.

3.4.1 Downloading Rules

Downloading rules are a particularly important feature of the Workshop System. Practicality demands that, at any given time, most of the software objects in a large software project reside in Global Database rather than in the Workshop process. The problem of selecting which objects are to be downloaded from the Global Database into the Workshop process is critically important. In a file-based system, the analogous question arises in the context of which files should be read into editor buffers. In a rich software database, the problem is magnified by the greatly increased variety and smaller grain-size of the informa-

tion being manipulated.

In SE-KRL, knowledge about what kinds of objects are of interest in a given situation is explicitly encoding in a DefDownloading rule. In this rule, a pattern is specified that describes the kinds of objects that should be downloaded. Whenever this pattern is satisfied, the Workshop System will generate a background request for the specified objects, thus allowing work to continue while the information is being retrieved from the Global Database.

An example of a very simple downloading rule would be :

```
(DefDownloading Personal-Objects
: test (AND (Is-Fact Workshop
              Current-Person ?Person)
            (Is-Fact ?Person
              Has-Interest-In ?Object))
: object ?Object)
```

3.4.2 Coordination and Abort Rules

Coordination rules are used to define knowledge about what changes are implied by other changes. A coordination rule consists of a pattern, which detects when the coordination should be applied, and a body which specifies what the implied change is.

Abort rules are used to specify what changes are illegal. An abort rule consists of a pattern which detects the illegal change, and an explanation which describes to the user why the change was illegal. Since abort rules have a lower priority than coordination rules, coordination rules can be used to "fix-up" user mistakes, and thereby circumvent a possible abort condition.

3.4.3 Invalidation Rules

Invalidation rules are structurally the same as abort rules, except they specify restrictions on modifications that can be made to the Global Database, rather than restrictions

on modifications in the Workshop process. A general premise of the Workshop System is that in his Workshop process, a user should be able to do anything he wants with information to which he has access. On the other hand, the project managers (and the project programmers) should be able to place strict requirements on the kinds of modifications than can be made to the Global Database. For example, in his Workshop process a programmer should be able to pick an arbitrary version of a system to be the "stable" version, but the same programmer should be able to count on a variety of conditions being true for any system marked as being "stable" in the Global Database.

An example of an invalidation rule would be :

```
(DefInvalidation Version-Not-Tested
: test (AND
  (Is-Fact ?System State Stable)
  (Is-Fact ?System Test-Suite ?Data)
  (NOT (Is-Fact ?System
    Tested-With ?Data)))
: explanation
  ("~A has not been tested with ~A."
  ?System ?Data))
```

3.4.4 Automation Rules

Automation rules are like coordination rules, except that the body of an automation rule can contain actions external to the Workshop System's database (such as sending mail or printing hardcopy). Since these actions cannot be retracted, they must be handled differently. In particular, automation rules have lower priority than abort rules since the Workshop System cannot in general cancel the effect of an automation rule that has already been activated.

3.4.5 Notification Rules

Notification rules allow the specification of arbitrary warnings, reminders, or memos.

They contain a pattern that specifies when the notification is appropriate and a message that contains what the notification should say. Unlike coordination and automation rules that guide the flow of information within or between computer processes, notification rules guide the flow of information to a human user.

4 A Software Engineering Knowledge Kernel

Although the Workshop System is designed to allow the addition of arbitrary software engineering knowledge, a significant amount of software engineering knowledge is required for the essential functionality provided by the Workshop System. Rather than bury this knowledge in the code of the Workshop System, this knowledge is specified in SE-KRL, and forms the kernel of all SE-KRL programs².

The main distinction between kernel knowledge and user supplied knowledge is that most kernel knowledge is read-only. As is only fitting, the rules specifying what parts of the SE-KRL kernel can be modified and how they can be modified, are themselves part of the SE-KRL kernel (one of which is a rule that specifies that the user cannot modify the rules that specify which parts of the SE-KRL kernel can be modified).

A full description of the SE-KRL kernel is not feasible in this article, but a comprehensive overview of the kernel classes will provide some insight into environment provided by the Workshop System.

4.1 Kernel Objects

The design goal for the SE-KRL kernel classes was to specify only those object classes

²analogous to a *standard prolog* found in languages such as ADA

that are essential to all software environments. The motivation for this was to avoid the clutter and confusion found in software environment database schemata that try to explicitly enumerate all conceivable software object classes. Although the costs of needless complexity are largely irrelevant for purely academic systems³, for practical systems the added power of each additional kernel class must be weighed against the cost to the user of added complexity - inappropriate knowledge is worse than no knowledge at all.

Therefore, in the Workshop System, what is not included in the kernel classes is just as important as what is included. To demonstrate the compactness of the kernel class specification, a complete inheritance tree is provided in Figure 1, with a more detailed description of most classes provided below.

4.1.1 Entities

Entity-Class is the root of the Workshop System class inheritance lattice. Every Workshop System software object is an Entity. All classes of objects directly implemented on the host machine are subclasses of Entity-Class. Examples of primitive classes are Number-Class, String-Class, and Symbol-Class (for Lisp machines). The only other subclass of Entity-Class is Object-Class, where an Object consists of a set of attributes (slot-value pairs).

4.1.2 Objects

The three subclasses of Object-Class are Process-Class, Description-Class, and Global-Object-Class.

A Process is an executable processes in the Workshop System. The subclasses of Process-Class correspond to the four kinds of processes in the Workshop System.

A Description is a database query. The list of all objects that have been downloaded from the Global Database and that match a query is dynamically maintained by the Workshop System.

A Global Object is an object that is stored in the Global Database. Note that Descriptions and Processes are not Global Objects in the default SE-KRL kernel, and therefore are not stored in the Global Database. This is done for pragmatic reasons, but is one of the aspects of the kernel that can be modified by the user.

4.1.3 Global Objects

The three subclasses of Global-Object-Class are Person-Class, Version-Class, and Job-Class.

A Person is an object that corresponds to a human. Every human that is relevant to a software project is instantiated as a Person object.

A Version is an object can exist in multiple versions. The main distinguishing attribute of a Version (other than its source definition) is its job-stamp (see below).

Jobs are particularly important objects in the Workshop System, because a job captures the notion of a human software process. Whenever a programmer or manager creates new software objects or modifies old software objects, he has some purpose in mind. In the Workshop System, each such mental construct is instantiated as a Job, and every software object is annotated with the Job that was current when that object was created.

The Jobs in a software project can then be used to help answer questions about why a change was made, information that is normally either lost during software development, or is very difficult to retrieve.

Another important function of Jobs is to distinguish one version of an object from another. Unlike most systems that use a

³at times, it even appears to be a goal

Figure 1: SE-KRL Kernel Class Inheritance

counter or some kind of date-stamp to distinguish versions, the Workshop System uses a *job-stamp* (the Job that produced a version) as the distinguishing characteristic. Experience has shown that this novel approach to version identification is a particularly valuable feature of the Workshop System.

4.1.4 Versions

The two subclasses of Version-Class are Module-Class and Program-Object-Class.

A Module is an aggregate object, that is used to structure a software system. An element of a Module can be any other Version

object; therefore, Modules can be both nested and overlapping.

A host system file is conceptually a very simple Module that contains a list of Programming Objects (see below). Various subclasses of Module-Class will be defined by the user to differentiate between the various kinds of software object groupings that are of interest to that user.

A Programming Object is an object that has source text that can be compiled or interpreted. The only subclass of Program-Object-Class in the kernel is SE-KRL-Class which is the superclass of all SE-KRL

language defining forms. The nine subclasses of SE-KRL-Class are Class-Class, Slot-Class, Operation-Class, Downloading-Class, Coordination-Class, Abort-Class, Invalidation-Class, Automation-Class, and Notification-Class, corresponding to the nine SE-KRL defining forms.

5 Conclusions

The ability to easily extend a software environment with knowledge about new software objects, software operations, and software rules, is an essential feature of a software environment that is intended to evolve as the science of software engineering evolves. An external database and a general inference engine combined with a knowledge representation language specifically designed for software engineering can effectively provide this capability.

The use of the software engineering knowledge representation language to specify the fixed kernel of the software environment is an effective way of presenting a consistent and comprehensible view to the user of the environment, as well as providing a guide for environment extension through additional knowledge. A particularly important element of this kernel are *job* objects, which capture the actual process used to evolve the software objects.

Finally, the combination of a knowledge-based software environment with an extensible object manager such as the Odin System, provides a complete software environment that is practical with current technology and that can be incrementally improved as knowledge-based technology improves.

6 Acknowledgement

A special thankyou to Lee Osterweil for supporting the decade of graduate work that

produced the Odin System. Thanks also to Bob Balzer and Tom Cheatham without whom Evolutionary Software, and therefore the Workshop System, would not exist.

References

- [1] *The ART User's Manual*. Inference Corporation, 5300 W. Century Blvd, LA, CA.
- [2] Robert M. Balzer. Living with the next generation operating system. In *Proc. 4th World Computer Conference*, September 1986.
- [3] Geoffrey M. Clemm. *The Odin System: An Object Manager for Extensible Software Environments*. Technical Report CU-CS-314-86, The University of Colorado, Boulder, CO 80309, February 1986.
- [4] Donald Cohen. Automatic compilation of logical specifications into efficient programs. In *Proc. 5th National Conference on Artificial Intelligence*, August 1986.
- [5] Leon J. Osterweil. The software process is software, too. In *Proc. 9th International Conference on Software Engineering*, 1987.