Integrating Configuration Management into a Generic Environment

Axel Mahler and *Andreas Lampen*
Technische Universität Berlin, West-Germany

Abstract

The software development process consists of a number of complex activities for work coordination, organization, communication, and disciplines that are essential for achieving quality software, maintaining system integrity, and keeping the software process manageable. Software Engineering Environments can be helpful instruments in pursuing these goals when they are integrated, open to extension, and capable of adapting to real processes as they occur in software development projects.

Adaptibility and the ability to perform adaptions *rapidly* are crucial features of SEEs. In this paper we are presenting an approach to rapid environment extension that provides the means to capture characteristics of software development processes and realize environment support for these processes by using existing tools. An object oriented *environment infrastructure* is the basis for achieving these goals while providing and maintaining an integrated behavior of the environment. The presented approach is demonstrated by defining a set of *classes* for version control and configuration management that model the behavior of an existing configuration management toolkit.

1. Introduction

The purpose of Software Engineering Environments (SEE) is to increase the efficiency of software production, and to enhance the quality of the developed software. The key concept in achieving this goal is a comprehensive support for the software development *process*.

In theory the effect of an environment would be to relieve the members of a project team from having to be aware of complex procedures and behavior protocols that are essential for the functioning of a project. The environment ensures automatically that these protocols are followed. Programmer productivity is increased because the programmer can better concentrate on the problem he/she is working on.

1.1. Software Engineering Environments must be adaptable

For some reason SEEs tend to not satisfy the expectations that are associated with them. Despite the recent "CASEmania" many integrated toolsystems and environments suffer from low acceptance or simply don't perform as expected. It has even been noted that environments eventually happen to be counterproductive.[1] Over the last decade substantial research has been (and still is) conducted to attack these problems and to generally improve the usefulness of SEEs.

While early SEEs were normally centered around a static model of the software development process (derived from a software development method) it has been learned that software development projects are highly dynamic processes, strongly dependent on the people involved, the size and complexity of the developed system, as well as external constraints such as customer relations. Today, it is understood that the actual process of developing software is more or less different for every project.[2] *SEEs must be able to cope with changing characteristics of the software development processes that have to be supported.*

What is needed is a customizable, highly flexible, and extensible SEE-frame/kernel. Some of the more obvious reasons for the necessity of SEE extensibility are:

- different projects need different kinds of support
- an environment should adapt to real (social) processes instead of forcing the teamwork structure to adapt to the environment
- fault-tolerance against environment design flaws
- support for environment evolution according to evolving requirements when a project proceeds (for example from implementation to maintenance phase)
- allowing *partial completeness* of an environment, implying delayed functional completion when suitable tools become available (rather than integrating insufficient tools "just to be complete")

During the last years, it has been widely accepted that good SEEs should be integrated, yet open and highly extensible.[3, 4, 5] Recent research has furthermore led to an increased awareness that in order to make the extension approach feasible, the extension of a SEE can be performed *rapidly*.

A good SEE shall combine the integrated look and feel of − for example − Smalltalk or Lisp systems with the general scope and applicability of tool collections as in the Unix environment. The initial establishment of a stable, basic environment infrastructure for a given project shall be a matter of *days*. To apply major changes/extensions (such as adding new classes of software objects to the system) shall be a matter of *hours*, and to apply minor changes (for example set user preferences, or fine tune certain object behavior) shall be a matter of *minutes*.

1.2. Recent approaches to environment integration

Many recent research projects have adopted the notion of *Software Object* as central for environment extension mechanisms.[6, 7, 8] The specifics of the software development process are captured by classifying the pieces of information evolving in a project, and describing related *behavior* of these information objects (we will refer to them as *software objects*).

The object oriented paradigm is used to specify external properties of software objects, and to provide for formally defined access protocols to be followed when manipulating these objects. Use of type inheritance makes it possible to specify general functionality on an abstract level (*abstract superclasses*), making the functionality available for use/reuse in application specific class definitions (*specialized classes*). The specified functionality is turned into action by corresponding *process programs*, formal (and executable) descriptions of particular software engineering activities, enacted by a process program interpreter. These process programs are the *methods* associated with a given software object class.[9, 10] Later in this paper, we will demonstrate how this technique is used to integrate such vital services as version control and configuration management into an environment, and how this functionality is (re-)used in specialized object classes.

Another interesting, yet entirely different, approach to environment integration has been taken with the Field environment.[11] Field integrates existing tools under a consistent graphical user-interface framework. Tools communicate via messages, representing information and commands intended for other tools. Messages are sent to a central environment message server which dispatches the messages to all tools possibly interested in them (*selective broadcasting*). The technique permits to integrate unrelated tools by adding a *message interface* to them. While this approach makes it possible to create moderately highly integrated environments from existing tools without a great deal of work, it requires some modifications to the tools' source code.

The most advanced approaches (for example [9]) are using knowledge based technology as main extension mechanism for SEEs. Today, we are moving towards a point where a SEE is basically a system that allows to store *knowledge* about how a particular project is working, and *functionality* that is intelligently applied according to the project specific work patterns stored in the knowledge base.

In the remainder of this paper we will briefly introduce the STONE project, a research project aiming at development of a generic SEE kernel especially suited for *environment education*, and outline STONE's approach to rapid tool integration. The subsequent description of the *shape toolkit*, a Unix-based set of integrated tools for version control and configuration management, marks one of the starting points for the STONE project. In the final section we will integrate the functionality of this toolkit into a STONE system environment.

2. The STONE Project

The work described in this paper is part of the STONE-project† aiming at development of a generic SEE kernel, open to extension and integration of new as well as existing tools. Some of the covered research areas, such as *object management*, *software process modeling*, and *user interfaces* are similar to those of the Arcadia project.[10] As environments must be adaptable to particular project structures, there must also be people, qualified to perform the adaptation. STONE's special agenda is support of environment *education*. Education will be supported in two principal ways:

1) providing a simple and well structured, yet highly functional environment kernel suitable for *education in construction* of software engineering environments, and
2) providing a basis for research and education in practical software engineering.

While the object oriented paradigm is more and more used to provide a formal framework for the specification of software process characteristics, it has rarely been described how these formalisms are used for actually integrating a complex tool suite into an environment. The purpose of this paper is to describe how the functionality of an existing toolkit for software configuration management (SCM) is integrated and made accessible in the STONE environment.

2.1. Rapid Tool Integration

Within the STONE project, a number of environment integration strategies are researched. Besides user-interface integration and fine-grain data-integration based on an object oriented data management facility, Ranft Rehm Uhl Sekerinski there is also research in *rapid tool integration* mechanisms. The main goal of this integration approach is to be able to very rapidly integrate existing Unix-tools into an integrated environment. The concept is based on soft-

† STONE − a STructured and OpeN Environment, a research project supported by the German Ministry of Research and Technology (BMFT) under grant ITS-8902E8

ware objects modeled after the file notion of Unix but which are based on a consistent, general purpose type system. The typed software object abstraction is built on top of a general attribution mechanism for Unix files and versions.

The type system allows to describe *classes* of software objects that are more than just files. For example, a C-language source object would not just be an ordinary file that happens to contain C source code but would be associated with properties (attributes and methods) that are characteristical for C code modules. The type system is defined in an object oriented specification language called *CHieF* (Class Hierarchy Definition Facility Wiegel 90), featuring multiple inheritance, generic classes, method overloading, dynamic identification, and easy schema modification. This specification mechanism allows to easily describe the properties of the various software objects encountered in the software development process, such as relations to other software objects, or dependencies. The specified methods for a given object class are implemented by Unix-tools or a combination of Unix-tools (shell scripts).

The type system is enacted by *OShell*, an object oriented command interpreter resembling the Unix Shell. OShell allows to send objects messages that trigger invocations of object specific methods. In that the system is not unlike Budd's *osh* idea described in. Budd Object Oriented Command Interpreter A similar concept is present in Odin's Clemm Odin Boulder 1986 request language, but lacks a sufficient inheritance mechanism.

The type system and the object shell allow to capture the external behavior of Unix tool processes, such as *vi*, *awk*, or *cc* if they are described as methods. Such behavior would for example be the creation of certain new objects (files) or the type of data that will be read from standard input or sent to standard output. This in turn makes it possible to construct type-safe *pipelines* within the object shell.

Adding support for a new kind of software object basically consists of defining the respective class, possibly using predefined functionality (such as version control) via the inheritance mechanism, and implementing the corresponding methods in form of calls to Unix tools (e.g. version control programs), or (O)Shell scripts that are also stored in the object base as special software objects of type Method. By applying this idea, we are able to amalgamate the prototyping power of Unix-typical tool combination (shell scripts, pipelines) with a formally integrated behavior of the environment.

While OShell itself is not a graphical user interface, it can be used to interact with a graphical environment *shell-tool*. In fact, the object shell represents an *abstraction* that has a well defined syntactical interface, suitable for formally communicating with a user interface process that presents the concepts of the object shell in a graphical way on a workstation.

With its loosely coupled tool integration concept and the rather simple way of dealing with object-base schema data, OShell is easily extensible for use in distributed environments. The flexibility that is known from using remote shells will directly be obtainable from a concept of remote OShell.

The concept of an object-oriented command interpreter provides the framework for integrating unrelated tools on top of a software object base into a moderately highly integrated SDE. The described work is unique in its consequent use of object oriented principles and techniques that are almost seamlessly combined with the well-known concepts of the UNIX environment.

STONE's generic environment kernel provides the basic technology to build comprehensive, highly integrated environments with reasonable effort. The simplicity, orthogonality, and expressiveness makes the described approach a good basis for environment education. The student (as well as the teacher) can concentrate on the *process* that shall be supported, rather than being concerned with the intricacies of adapting any existing, closed, special purpose SDEs as used in the commercial environment.

A schematic overview about the architecture of the rapid tool integration facility is given in Fig. 2.1. A more detailed description of OShell and the CHieF typesystem is beyond the scope of this paper.

3. The Shape Toolkit

Shapetools is a collection of programs for version control and configuration management adhering to the Unix toolbox philosophy. It is basically a traditional toolset that supports general functions for a particular problem area but doesn't care about the particular needs of individual projects and therefore must be customized for non-trivial project settings.

The toolkit consists of a set of version control commands and *shape*, a configuration building tool similar to Make. Feldman Make A Program *shape* and the version control commands are integrated on top of AFS (*Attributed File System*), a dedicated version object base (AFS is an internal name and should not be confused with the Andrew File System). The system features a configuration identification and -build process that has full access to all revisions in the object base (other than make which only knows about plain files). AFS also supports derived object management, i.e. it maintains a cache of multiple versions of compiled object-files (for example compiled c-files with different compile switches).

The *Shapefile*, shape's system description file, uses Makefile-style dependencies as (versionless) *abstract system model* and employs configuration selection rules to dynamically bind unique object versions in the object base to the names listed in the system model. The version selection mechanism exploits AFS' ability to maintain any number of arbitrary object attributes. This yields a configuration identification mechanism similar to DSEE's *configuration threads*. Leblang Chase

One of the most useful features of shape is its support of various variant administration techniques. Shape makes no assumptions about the semantics of the *variant* notion other than having *alternative (conceptually equivalent) instances of the same concept* (e.g. *module*). Despite the particular semantics of application specific variant-concepts, there is a small number of techniques to physically handle variants. These techniques are what shape supports. Most commonly used techniques are:

- equally named files in different directories
- one source file representing multiple variants that are extracted by a preprocessor using different preprocessor switches (e.g. conditional compilation)
- one source file processed by different tools or different tool versions/variants (e.g. cross compilation, different coders)
- combinations of the above.

Shape includes a variant definition facility that allows flexible, and convenient handling of all of the above variant administration techniques.

While the version control system is straightforward to use, it can be tricky to make ultimate use of *shape*. Besides lots of options and switches, *Shapefiles* aren't as easy to write as Makefiles. To help with this problem, we have introduced an include-mechanism into shape. The idea is that parts of the Shapefile that could be shared among all developers (such as version selection rules or variant definitions that are tricky to write but rarely need to be modified — if well designed) can be written once for a particular project, and be placed in a standard-location from where they can be included in the individual developers' Shapefiles. This technique not only facilitates Shapefile-writing but also helps to implement project-wide standards for structuring the Shapefiles.

We used this technique for setting up a teamwork support environment to manage development, releases, and maintenance of the shape toolkit itself. Although the evolving release management system proved to be *very* useful for implementing project wide software management standards, and automatization of the tedious job of creating system distributions, it became clear that a much more systematic way to describe a particular project setup is needed. Pralle Driving The use of object oriented techniques to define software process object properties and a corresponding command language appeared very appealing. Using such a model, it would for example be possible to call for the creation of a *system model object* as system management handle, and to define functions that are now defined as standardtargets as *methods* of a class SystemModel.

In the following section we will describe how project wide configuration management policies can be consistently enforced by the environment while making it actually much easier to use (and adhere to) these principles without using red tape. Special care has been taken to retain the flexibility of the shape program.

4. A Class System for Version Control and Configuration Management

This section describes the integration of support for version control and configuration management into the STONE environment. It is our goal to provide at least as much functionality as does the shape–toolkit. Version control and configuration management functions are highly relevant for nearly all software objects evolving during the development of a software system. Nevertheless, integration of these functions into a generic environment (like STONE) should not impose any restrictions to the integration of further parts into the environment. The work described in this section can also be seen as a case study to test how capable the rapid integration mechanism of STONE is.

As described earlier, our work will mostly consist of defining appropriate classes with corresponding methods and attributes. Due to limited space, we cannot provide a complete formal definition of each class in this paper. We will rather give a sketch of a class hierarchy (see figure 4.1) that contains the essential classes for version and configuration control. We will describe most classes informally. Just for a few of them, we will give a formal CHieF definition containing the most important attributes and methods.

The presented class hierarchy shall be capable of capturing characteristics of the software development process and tie them to existing tool processes in UNIX. In our model we define a set of *foundation* classes. Foundation classes are abstract classes that are not intended to have instances. Their only purpose is defining abstract functionality corresponding to general procedures of the software development process. *Specialized* classes, classes that have instances in the software developers workspace, are derived from the foundation classed by using multiple inheritance.

4.1. The foundation classes

The root class Object describes the basic properties shared by all objects. This implies, that all objects have a network wide *unique identifier*. Unique identifiers can be used to realize references from one object to another. With Object we also define a create and a destroy method as default for all derived classes.

The edges in the graph in figure 4.1 display the *inherits–to* relation. So, all classes in the graph inherit the unique identifier and the default create and destroy method.

Labeled objects are objects that have a visible representation at the user interface. They have a name attribute, and – for use in graphical interfaces – an icon. Labeled objects are objects that the user can directly send messages to. They roughly correspond to (UNIX) files. A typical property of Labeled objects is the location in a hierarchically structured namespace (directory tree).

Source objects are all objects, that are produced/manipulated directly by a human. They share a method edit which allows a human to manipulate the contents of the source object. The method browse serves for showing the contents of a source object in a human readable representation. Source is an abstract class that only serves for deriving specialized source classes from it. Specialized source classes will typically rely on redefinitions of the edit and browse method. A source class like eg. AsciiText can be manipulated with a normal text editor while for example LineArtGraphics requires a special draw tool as editor.

```
class Source
  {
    inherits: Labeled;
    features:
      USER author;
      TIME  lastModified;
     virtual edit ()
        { ... }
     virtual browse ()
        { ... }
  }
```

The class Derived represents objects that are all objects generated automatically from another object by any method execution. A closer look at this class can be found by the end of this section.

To illustrate the use of the type concept, we introduce the class CSource, as a typical source class, into our class hierarchy. CSource objects contain program source written in C.

```
class CSource
  {
    inherits: Source;
    features:
      INT status; // open (1)
                  // erroneous (2)
                  // validated (3)
      validate ()
        { ... }
      edit
        {
         command:
           "emacs $self"
        }
      browse
        {
         command:
           "more $self"
        }
  }
```

4.2. Controlling revisions of source objects

In a STONE based SEE, a developer shall be able to keep track of source object revisions representing different stages of development. Revisions of source objects are created explicitly by the user when the status of a workpiece shall for any reasons be saved for later backup. The design of the functionality of the revision management is oriented at the shape–toolkit's version control system. It includes automatic revision numbering, a state model for source revisions, and symbolic names for single revisions.

The concept of the class History is to offer revision control capabilities on an abstract level, intended for combination with specialized Source class behavior in order to add a "memory" to a source class. SourceHistory objects are able to *remember* their state at certain points in time (method save). These remembered states correspond to *revisions* in traditional version control systems. Memorized states of source objects are not themselves objects but part of the internal state of the corresponding SourceHistory object. History objects have the capability of mimicing a source object in all of its remembered states. This is achieved by combining a SourceHistory object with a defaultVersion attribute.

History is a generic class that is parameterizable by any source class. The following box shows the (incomplete) definition of the History class in CHieF.

```
class History [T]
      Source T
   {
      inherits: Labeled;
      features:
         VERSIONID: defaultVersion;
         T : tmp;
         create (T)              { ... }
         save ()               { ... }
         bindToVersion (SelectionRule)
            { ... }
         recall (VERSIONID)
            {
              yields T;
              command:
                 "vcat -V$1 $self > $tmp"
            }
   }
```

It is always possible, to *clone* (method recall) new objects of a certain Source class from object versions. Such a cloned object would have the type of the source class that was combined with History by the inheritance mechanism.

The class CSourceHistory is an example for a specialized class, the user deals with. It is constructed from History and CSource by using multiple inheritance. From the user's point of view, instances of the class CSourceHistory behave like CSource objects but additionally are able to maintain a history of revisions. When an instance of the class CSourceHistory receives the message browse, which is inherited from CSource it looks up the actual defaultVersion (inherited from History) in order to select the revision that acts as baseline for the edit method.

The browse method, like most other methods inherited from the corresponding source object class has to be redefined by adding a preceding revision addressing. According to the dafault version, browse recalls the appropriate revision and applies the method browse of the parent source class to the recalled revision. This is illustrated in the following example. Remember that tmp is an instance of class CSource (defined in the History class, so that browse is not recursive.

```
class CSourceHistory
   {
      inherits: CSource,
               History [CSource];
      features:
         browse ()
            {
              command:
                 "recall (defaultVersion);
                  tmp.browse ();
                  tmp.destroy ()"
            }
         validate ()
            { ... }
   }
```

The actual default version of a History object does not always need to be defined statically. It may be made dynamically on each execution of a method that needs an actual default version. This idea will be discussed in more detail in the section that deals with work contexts.

4.3. Configuring Systems

The task of configuring systems splits into two subtasks.

- Identification of all components and component versions that go into the system and
- the actual build process of performing the necessary transformations.

When looking at the structure of the Shapefile, the information needed for system building divides into three parts. These three parts – system model, version selection rules, variant definitions – correspond to three classes in the new model.

The system model describes an acyclic graph of dependencies between all components the system consists of. Nodes in the graph representing components that do not depend on any other component, represent source objects of the system. All other nodes represent derived objects. Derived objects are produced automatically from the (source or derived) objects, they depend on. With each dependency in the system model, a derive method describing the necessary derivation, is associated.

Instances of class SystemModel will act as representants of a system and take the place of this part of the Shapefile. The syntax for describing the dependencies is basically the same as in shape (and make), where dependencies can either be expressed explicitly or implicitly by giving generic dependencies. In make/ shape, generic dependencies (single/ double suffix rules) rely on filename suffixes that should express the type of the files involved. In our model, generic dependencies base on object types, which provide much more reliable information about the objects.

Going further than the scope of the system model part in a normal Shapefile, a SystemModel object will also define the source components associated with the system it describes. This is necessary, because not all source objects will have a corresponding node in the dependency graph. These are for example the SystemModel object itself, manuals, or documentation.

The method establish of SystemModel generates an initial object for each node in the dependency graph as far as it does not exist yet. This must happen at least once for each system, before the first build message can be sent. We will revisit this issue in the following section about the process of system building.

The class corresponding the the version selection part in the Shapefile is SelectionRule. SelectionRule objects drive the default version in a History object. This is the selection of a specified version within a History object when it receives a message for the managed source object. The contents of a SelectionRule object is a regular version selection rule like in shape. Selection rules in shape are based on expressions describing a set of required attributes, the version to be selected has to have.

An important advantage of having selection rules as autonomous objects (rather than being just another part in a system description file) is the ease of sharing selection rules between different developers and projects. According to our experience with shape, selection rules need not to be changed very frequently and it is very likely that a standard set of selection rules can be used for most developments. Specially customized SelectionRule objects containing specific rules, for example rules based on a single unique attribute like version number of symbolic name, can be created quickly when needed.

The variant definition part in a Shapefile has its counterpart in the Variant class. As for version selection rules, we can easily adopt the shape syntax for the contents of objects of class Variant. So a Variant object contains the description of a *variant family* that has multiple targets corresponding to different variants of the environment. A shape variant class might eg. be *interface language* with the targets *German, English* and *French*. Each target describes the variant properties of the environment by means of attribute definitions interpreted (used) by the build process.

Typically, variant definitions should be made for a whole project, as a means to anticipate the evolution of the system to be capable to run in a predefined set of target environments. Experience with shape showed, that the variant mechanism is used extensively. Variant families address porting systems to different environments, use of different tools, and controlling the derivation process for example to generate compiler output with different quality for eg. debug or release purposes. Especially the latter two cases led to a default variant raster, that can be used with only small modifications in most projects.

With the classes defined above, we are able to produce a SCM environment that systematically enforces the conventions we invented for the usage of shape in cooperative work (see section 3.1). With respect to version selection rules and variant definitions, we have a more flexible and reliable mechanism than that of include files.

4.4. Work contexts

Instances of class WorkContext can be uses to associate a special version selection rule (a SelectionRule object) and a variant definition (Variant object) with the actions a user performs. As discussed earlier, History objects will be able to perform dynamic version binding by means of work contexts. This concept roughly corresponds to shape. During the execution of shape, there is always an *current* version selection rule and *current* variant definitions.

4.5. The build process

The method establish of SystemModel objects constructs an object structure as described in the system models dependency graph in the real object world. It does this by creating an initial object for each node in the dependency graph that does not already exist. Object relations such as dependencies (the edges in the graph) are expressed by references. Sending the message build to any derived object constructed by establish, will cause the object to derive/update itself from the objects it depends on. If the derived object already exists, the build message causes the object to update itself if necessary. In this section we will have a closer look on what happens during the system build process.

During system model establishment, each Derived object will be associated with an individual derive method. The derive method is an autonomous object of class Method

After being established, a system or a part of a system can be built by sending the message build to any derived object corresponding to a node in the dependency graph. The build method will first recursively send a message build to all objects the actual object depends on. If any of the dependents has been updated, the actual objects needs to be newly produced. If the derived object depends on a source object, it is newly produced if the corresponding source object has changed since the derived object had been updated the last time. The algorithm described above is basically the make/shape algorithm.

The production (or updating) of a derived object is driven by the method object, associated with the derived object during establishment of the system model. Additionally, the work context, in especially the variant definition has an effect on the production process. For example if the work context is designed to set the surroundings for testing a system, it may refer to a variant named *debug* that causes the compiler to place debugger information into the executable code and skip any code optimization. In this case, for the actual production a new Method object has to be generated, that describes the exact produce method for the derived object in this special work context. This new Method object will be derived from the method object already associated with the derived object.

4.6. A closer look at derived objects

Due to different production methods, it is possible to produce different variants of a derived object from one source. Similar to History objects, Derived objects consist of a set of versions but behave like a singe version. Although conceptually not necessary, we will in this section make a difference between derived objects produced directly from a source object and derived objects produced from a set of other derived objects.

In a derived object produced from a source history, different versions evolve either by derivation from source versions, or by using different production methods. Both, the version selection and the generation of the production method are driven by the actual work context. In order to be able to identify, which source version the derived version was made from and how it was produced, it is attributed with the version binding and the actual variant definitions that were current during production.

Derived objects produced from other derived objects (eg. a program linked together from several .o files) must also be attributed in order to be able to reconstruct which versions they evolved from and in which context they were produced. They have a set of references to all objects (versions of derived objects) they were generated from. Additionally, the method object that has been produced for this special derivation has to be preserved.

5. Conclusion and status of implementation

Did we achieve our goal to smoothly integrate configuration management functionality into a generic environment ? In the current state of development, we cannot yet answer this question ultimately. Although it is quite clear, how the presented concept will be implemented by using CHieF and OShell, experiences with the day to day use will bring up a lot of fine tuning, making the system really usable.

The CHieF compiler and the access routines for the class directory are ready implemented. The first implementation of the OShell is scheduled to be finished in July. So we plan to evaluate our concepts in the real software development world during July/August this year.

References

1. Peter F. Elzer, "Management von Softwareprojekten," *Informatik Spektrum*, vol. 12,4, pp. 181-198, Springer Verlag, Berlin, August 1989.
2. Vic Stenning, "On the Role of an Environment," *Proceedings of the 9th International Conference on Software Engineering*, pp. 30-34, IEEE, Monterey, California, March 1987.
3. A. Nico Haberman and David Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering*, vol. 12, 12, pp. 1117-1128, December 1986.
4. G. Snelting, "Experiences with the PSG − Programming System Generator," *Lecture Notes in Computer Science*, vol. 186, 2, pp. 148-162, Springer Verlag, Berlin, March 1985.
5. A. Goldberg, *Smalltalk-80, The Interactive Programming Environment,* Addison Wesley Publ. Company, Reading, Menlo Park, London, Amsterdam, 1984.
6. Walter F. Tichy, "Tools for Software Configuration Management," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1-20, German Chapter of the ACM, Grassau, FRG, January 1988.
7. Geoffrey M. Clemm, "The Odin System: An Object Manager for Extensible Software Environments," *CU-CS-314-86*, The University of Colorado, Boulder, Colorado, February 1986.
8. William Courington, "The Network Software Environment," *A Sun Technical Report*, Sun Microsystems, Inc., Mountain View, CA., May 1989.
9. Geoffrey M. Clemm, "The Workshop System − A Practical Knowledge-Based Software Environment," *Software Engineering Notes, Vol. 13, No. 5*, pp. 55-64, ACM Press, Boston, Mass., November 1988.
10. Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil, and Alex L. Wolf, "Foundations for the Arcadia Environment Architecture," *Software Engineering Notes, Vol. 13, No. 5*, pp. 1-13, ACM Press, Boston, Mass., November 1988.
11. Steven P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, vol. 7, no. 4, pp. 57-66, IEEE Computer Society, July 1990.