Integrating Configuration Management into a Generic Environment

Axel Mahler and *Andreas Lampen*
axel@coma.cs.tu-berlin.de, andy@coma.cs.tu-berlin.de
Technische Universität Berlin
Sekr. FR 5-6
Franklinstraße 28/29
D–1000 Berlin 10, 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.

Adaptability and the ability to perform adaptations *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[6]. 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[16]. *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[8, 9, 15]. 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[2, 4, 18]. 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[3, 17]. 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[14]. 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 [3]) 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[17]. 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.

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

2.1. Rapid Tool Integration

Within the STONE project, a number of environment integration strategies are researched. Besides user-interface integration[5] and fine-grain data-integration based on an object oriented data management facility[19], 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 software 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[10].

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[20]), 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[1]. A similar concept is present in Odin's [2] 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.

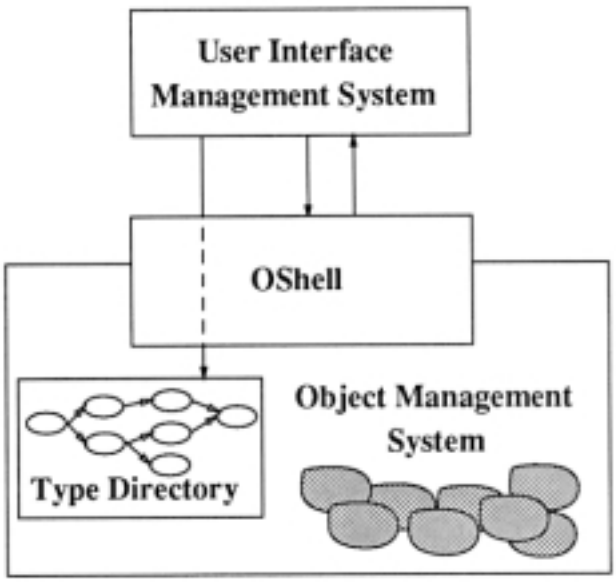

Fig. 2.1. Rapid tool integration in STONE

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 SEE. 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 SEEs as used in the commercial environment.

3. The Shape Toolkit

Shapetools[12] 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[7]. Shape and the version control commands are integrated on top of AtFS (*Attributed File System*), a dedicated version object base. 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). AtFS 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 AtFS' ability to maintain any number of arbitrary object attributes. This yields a configuration identification mechanism similar to DSEE's *configuration threads*[11].

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[13].

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 with well defined properties. Functions that are now defined as standard targets will then be *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. 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 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 tool 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 CHieF like definition containing the most important attributes and

methods. For reasons of comprehensiveness, we will not stick strictly to the CHieF syntax.

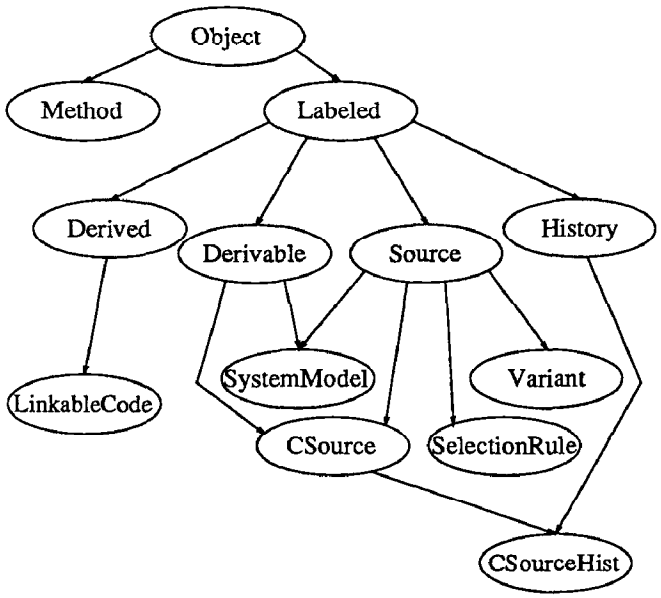

Fig. 4.1. The Class Hierarchy

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 and *specialized* 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 classes 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 name space (directory tree).

Source objects are all objects, that are produced and manipulated directly by a human. They share a method edit which allows the developer 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. At the abstract level of Source, edit and browse are virtual methods without concrete implementation (tool binding). Specialized source classes rely on redefinition of the edit and browse methods. 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 CSource is a typical specialized source class derived from Source. Objects of class CSource contain program source code written in C.

```
class CSource
  {
    inherits: Source;
    features:
      INT status;
      validate ()
        {
          command: "lint $self"
        }
      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. Each History object carrys the attribute currentVersion that points to the remembered state to be addressed when receiving a message.

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 [SOURCETYPE]
      Source SOURCETYPE
  {
     inherits: Labeled;
     features:
        VERSIONID: currentVersion;
        SOURCETYPE: tmpSrcObject;
        save ()
           {
              command: "save $self"
           }
        bindToVersion (SelectionRule: rule)
           { ... }
        recall (VERSIONID: vid)
           {
              yields SOURCETYPE;
              command:
                 "vcat -V$vid $self > $tmpSrcObject"
           }
  }
```

The method bindToVersion interprets a given SelectionRule object in order to get a new version binding (version number) to be set as currentVersion. Up to now, evaluation of selection rules is internal functionality of the shape program rather than being an own, callable tool. Insofar the implementation of bindToVersion is quite complicated (a special Shapefile has to be constructed) and we left it out here.

It is always possible, to *clone* (method recall) new objects of a certain Source class from object versions. Such a cloned object has the type of the source class that was combined with History by the parameterizing 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 edit (inherited from CSource) it looks up the actual currentVersion (inherited from History) in order to select the revision that acts as baseline for the edit method.

```
class CSourceHistory
  {
     inherits: CSource,
               History [CSource];
     features:
        browse ()
           {
              command: †
                 "recall $self, -V$currentVersion;
                  browse $tmpSrcObject;
                  destroy $tmpSrcObject"
           }
  }
```

The browse method, like most other methods inherited from the corresponding source object class has to be redefined by adding a preceding revision reconstruction. According to the current 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 previous example. Remember that tmpSrcObject is an instance of class CSource, so that browse is not recursive.

4.3. Configuring Systems

The process of configuring systems splits into two tasks.

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

In the shape approach, all information about how to configure a system is concentrated in the Shapefile. The information stored in the Shapefile divides into three parts. These three parts – system model, version selection rules, and variant definitions – correspond to three classes in our

† The command body in this example is written in OShell command syntax which is basically:
`<message> <recipients> , <arguments>`

new model. The system model part of a Shapefile describes an acyclic graph of dependencies between the components, the system consists of. Nodes in the graph representing components that do not depend on any other component, must be source objects. All other nodes represent derived objects. Derived objects can be produced automatically from the (source or derived) objects, they depend on. With each dependency in the system model, a script describing the necessary derivation, may be associated.

Instances of class SystemModel will act as representatives of a system and take the place of the system model 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 indicate 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 usual Shapefile (requiring only the dependency graph) a SystemModel object will also define a list of all source components and a list of all derived components associated with the system it describes. Normally, not all source objects have a corresponding node in the dependency graph (for example manuals and documentation). We will discuss the role of SystemModel objects more deeply in the chapter about the system build process.

4.3.1. Version Identification

The class corresponding to the version selection part in a Shapefile is SelectionRule. SelectionRule objects drive the version selection in a History object by setting the currentVersion attribute before sending any messages to it. 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 experiences 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* or *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 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). With respect to version selection rules and variant definitions, we have a more flexible and reliable mechanism than that of include files.

4.3.2. The build process

Building a complex software system usually requires a certain number of derivations to be performed. In this context, we introduce the classes Derivable and Derived into our class hierarchy. A Derivable object can be automatically transformed into a Derived. Each Derived object is associated with the Derivable it was constructed from. When a Derivable object is modified, the corresponding Derived needs to be updated. For this purpose, the Derived object carries a method build, suitable to create a current instance of itself by performing a transformation of the corresponding Derivable object.

At the first glance, the class Derivable seems to be similar to Source. In fact, not each Source object is a Derivable. For example shell scripts or other interpreted program text need not to be compiled. Other examples are objects of the classes SelectionRules and Variants as presented before. Additionally, a Derivable needs not be a Source. In this case it is an intermediate derived, i.e. it is both Derivable and Derived. An example for this is C code generated automatically as intermediate code from any other language (eg. yacc, Eiffel, C++).

Each Derivable object is associated with a *system model*, either an *implicit* or an *explicit* one. A system model describes dependencies between Derived and Derivable(s), and the tools to be invoked for performing the derivation. This information is necessary for properly creating corresponding Derived objects together witch upcoming Derivable objects.

An *implicit system model* is comparable to a generic dependency rule known from shape. It describes simple

dependencies and derivation processes on *type level* without referring certain objects. This is typically a Source – Derived dependency like for example CSource – LinkableCode. Implicit system models can be (but are not necessarily) represented by objects of class SystemModel. On creation of a Derivable object associated with an implicit system model, the corresponding Derived will automatically be created too.

Usually SystemModel objects contain an *explicit system model*, describing a (more or less) complex system built from a number of components. Dependencies described in an explicit system model normally are associated with derivations transforming multiple components to one (or more) Derived. Mostly the components are themselves derived objects.

Typically, an explicit system model is needed when the system consists of an executable object that is to be constructed by linking together a number of LinkableCode objects. Furthermore, the same system model may contain a description on how to produce a "ready–to–distribute" archive file (eg. a tar file) from all sources. Speaking in terms of make/shape, a SystemModel may contain multiple *targets*. These are represented by multiple Derived objects in our new model. In the case of explicit system models, the SystemModel object takes the place of the Derivable object, the Derived objects mentioned above are constructed from.

SystemModel objects have a method establish, that creates all anticipated Derived objects in an initial state, ready to receive the build message. Remember, for objects associated with implicit system models establishing the derived objects is done automatically on creation. This is not feasible for explicit system models. Due to modifications of the system model, the number of Derived objects to be produced may change. In this case, subsequent activations of the establish method are necessary to carry over the changes to the system model to the real object world.

On execution of the build method of Derived objects derived from a SystemModel, build messages are sent recursively to all objects the Derived depends on. The real derivation (eg. linking all LinkableCodes) is performed after this. During the build process, unnecessary derivations are avoided. A system is rebuilt only if one of the derived objects it depends on, was newly created. A derived object depending on a source object, is newly produced if the corresponding source object was modified since the derived object had been built the last time. This algorithm is basically the shape/make algorithm.

Similarly to Historys, Derived objects maintain a number of versions, or better *virtual objects*. These virtual objects are different representations of (conceptually) the same derived object. Virtual derived objects come into being by modifying certain parameters of the derivation process. Inspired by the variant handling mechanism of shape, the build method of a Derivable is merely a method prototype containing macros to be expanded by arguments passed together with the method activation. To give an example, a LinkableCode object generated by a cross compiler with multiple back ends may contain target code for different machines (VAXen, MIPS or 680XX) depending on the an argument passed to the compiler. Each virtual object is associated with a description of the derivation process context that was in effect when it was built. Additionally, each virtual object carrys information about the *version* of the Derivable it was built from.

Derived objects produced from other derived objects (eg. a program linked together from several LinkableCode objects) also get marked, in order to be able to reconstruct which versions they evolved from and in which context they were produced. This has the form of a set of references to all objects (versions of derived objects) they were generated from.

Derived objects are maintained as a cache with limited size. Due to the potentially large number of different virtual objects (the cross product of all variant families multiplied with the number of versions), only a certain number of versions are stored. When the limit is reached, the oldest virtual object will be deleted when introducing a new one.

5. Conclusion

Get a box full of useful tools and some kind of *glue* to put in between the tools. Stick the tools together and put a smooth surface on top of the whole thing. This is the basic idea of the mechanism for constructing integrated software engineering environments, presented in this paper. Does this really work ?

To prove this, we tried to smoothly integrate an existing complex toolset supporting version and configuration management into our generic environment. The concept looks fine, as always! But we are sure that the ongoing implementation will uncover weaknesses. Although it is quite clear, how the presented concept will be implemented by using CHieF and OShell, experiences with day to day use have to bring up a lot of fine tuning, making the system really usable.

A first prototype of the CHieF compiler and the access routines for the class directory are implemented. OShell and a new version of AtFS (featuring unique identifiers and network support) are currently under construction.

References

1. Timothy A. Budd, "The Design of an Object Oriented Command Interpreter," *Software – Practice and Experience*, vol. 19, no. 1, pp. 35-51, January 1989.
2. 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.
3. 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.
4. William Courington, "The Network Software Environment," *A Sun Technical Report*, Sun Microsystems, Inc., Mountain View, CA., May 1989.
5. D. Eckardt, W. Hübner, and G. Lux-Mülders, "Konzeption der STONE–Benutzungsoberfläche THESEUS++," *STONE Technical Report ZGDV.006.1*, Zentrum für Graphische Datenverarbeitung, Darmstadt, December 1989.
6. Peter F. Elzer, "Management von Softwareprojekten," *Informatik Spektrum*, vol. 12, no. 4, pp. 181-198, Springer Verlag, Berlin, August 1989.
7. Stuart I. Feldman, "MAKE - A Program for Maintaining Computer Programs," *Software - Practice and Experience*, vol. 9, no. 3, pp. 255-265, March 1979.
8. A. Goldberg, *Smalltalk-80. The Interactive Programming Environment*, Addison Wesley Publ. Company, Reading, Menlo Park, London, Amsterdam, 1984.
9. A. Nico Haberman and David Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering*, vol. 12, no. 12, pp. 1117-1128, December 1986.
10. Andreas Lampen and Axel Mahler, "An Object Base for Attributed Software Objects," *Proceedings of the Fall 1988 EUUG Conference*, pp. 95-106, European Unix systems User Group, Lisbon, Portugal, October 1988.
11. David B. Leblang and Robert P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment," *SIGPLAN Notices*, vol. 19, no. 5, pp. 104-113, ACM, Pittsburgh, PA., April 1984.
12. Axel Mahler and Andreas Lampen, "An Integrated Toolset for Engineering Software Configurations," *Software Engineering Notes, Vol. 13, No. 5*, pp. 191-200, ACM Press, Boston, Mass., November 1988.
13. Ulrich Pralle, "Driving the Software Release Process with Shape," *Proceedings of the Fall 1990 EUUG Conference (submitted to EUUG conference)*, pp. 0-0, European Unix systems User Group, Nice, France, October 1990.
14. 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.
15. G. Snelting, "Experiences with the PSG – Programming System Generator," *Lecture Notes in Computer Science*, vol. 186, no. 2, pp. 148-162, Springer Verlag, Berlin, March 1985.
16. 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.
17. 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.
18. 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.
19. Jürgen Uhl, Bernhard Schiefer, Emil Sekerinski, Simone Rehm, Thomas Raupp, Michael Ranft, Richard Längle, and Karol Abramowicz, "The Object Management System of STONE - SOS Release 1.0 -," *STONE Technical Report FZI.001.2*, Forschungszentrum Informatik, Karlsruhe, April 1990.
20. Burkhard M. Wiegel, "Entwicklung eines Klassensystems für attributierbare Softwareobjekte," *Diplomarbeit*, Technische Universität Berlin, Institut für angewandte Informatik, Berlin, Februar 1990.