Jasmine: A Software System Modelling Facility

Keith Marzullo
Xerox Information Systems Division

Douglas Wiebe
University of Washington

ABSTRACT

Jasmine is a programming-in-the-large system designed by the authors at Xerox Information Systems Division. Jasmine consists of workstation tools and network services that help programmers develop, release, and maintain large software systems. Jasmine has three primary parts: (1) *system models* that describe the structure and versions of software, (2) context-relative, distributed naming of software components (supporting replication), and (3) a collection of tools that use (1) and (2) to manipulate software systems. We present an overview of these parts of Jasmine.

1. INTRODUCTION

Jasmine is a distributed, production quality programming-in-the-large system developed by the authors for use by software developers at Xerox Information Systems Division. We consider programming-in-the-large to consist of two parts: (1) the description of software systems by *system models*, and (2) the manipulation of software based on the information in system models. Jasmine system models describe software using a simple but powerful algebra based on sets and functions. Jasmine tools perform software manipulations such as copying and archiving files, compiling sources, generating releases, and browsing.

Current addresses: Keith Marzullo, Department of Computer Science, Cornell University, Ithaca, NY 14853 and Doug Wiebe, Department of Computer Science, University of Washington, Seattle, WA 98195.

1.1 Related Systems

Jasmine shares features with other programming-in-the-large systems. For example, Jasmine uses a hierarchical module interconnection language, as do Tichy [Ti80] and the Cedar System Modeller [Sc82]. Jasmine's method of referencing versions of system components resembles the approach used by the DSEE Configuration Manager [LC84]. Jasmine most closely resembles systems designed to be used in an actual production environment, such as [Fe79, Le83, LC84].

1.2 Jasmine Background

Like the Cedar System Modeller, Jasmine is based on experience with the DF software developed at Xerox PARC [Sc82]. The DF software has been used by the product divisions of Xerox to control software systems containing thousands of modules [Le83]. Inadequacies and inefficiencies in the DF software led to the development of Jasmine.

Jasmine tools have been developed for the Xerox Development Environment (XDE), a programming support environment that runs on Xerox personal workstations [Sw85]. Although XDE is written in the Mesa programming language [M*79], Jasmine makes no assumptions that the software it models is Mesa code.

Jasmine tools communicate with Jasmine network services via Courier remote procedure calls [Xe81]. At the present time, Courier depends on the Xerox Network Services (XNS) communications protocol family [Xe85] and hence is implemented only on systems that support XNS (this includes Berkeley 4.3BSD Unix). Efforts are currently underway to implement Courier on other communications protocols such as TCP/IP and ISO [Ar86]. Jasmine tools and services should then be portable to a wide variety of systems.

2. JASMINE SYSTEM MODELS

Programming-in-the-large systems usually maintain a database of information that describes the software being managed. For example, the well-known Unix program *Make* [Fe79] reads as input *makefiles* that describe the components of a system, the dependencies between components, and the shell commands needed to rebuild the system. Other descriptive mechanisms include functional languages [Sc82], attributed directories [Th83, KH83], attributed graphs [K*85], descriptive data objects [GB80], and special-purpose databases [C*85, ZW85, TL85, Es85].

The description of software systems by mechanisms such as these is now commonly referred to as *system modelling* [La80]. The elements of the descriptive database (be they programs, objects, graphs, or attributes) are called *system models*. A programming-in-the-large system generates and stores system models as software is developed. Development tools access the system models to perform manipulations on the software such as copying and archiving files, compiling sources, generating releases, and browsing.

Jasmine system models consist of data objects called *templates* and *images* that are grouped into *families* and evaluated in *contexts*. The next section describes the kinds of information that are represented in Jasmine system models. Following sections explain Jasmine system models in detail.

2.1 Categories of System Model Information

Four categories of information are represented in Jasmine system models: (1) relations between system components, (2) version information, (3) construction rules, and (4) verification rules.

(1) *System relations*. Three kinds of relations between software system components are represented in Jasmine system models. The first kind is the *modular decomposition* of a system into a hierarchy of subsystems and modules [Pa72]. The decomposition relation can be represented by a directed acyclic graph.

Second is the general class of *dependency relations*. For example, a *build dependency* is a partial ordering of the modules of a system, indicating precedence in the order in which modules can be compiled and linked to generate an executable form of the system. As another example, there might be a dependency between sections of the user's manual for a system and the source code for the public interfaces of the system (so that the manual gets updated when an interface changes). Dependency relations can be represented by directed graphs.

A third kind of system relation is the grouping of modules based on properties associated with the modules. For instance, the exported interfaces of a system may be grouped together for public release. In this case, the group contains those modules that have the properties of being interfaces and being exported. Module groupings may be represented by sets.

(2) *Version information*. The system relations above describe the structure of a particular version of a system. As a system evolves, new versions of modules and subsystems are generated. *Version families* describe the succession of such versions.

At times, *parallel development* may occur. For example, new features may be added to a software package at the same time debugging edits are made to the original version of the package. Thus development *forks* from the original package to the enhanced and debugged versions. Eventually the parallel development lines may be *merged* into a single version. Version histories can be represented by directed acyclic graphs.

(3) *Construction rules*. Operations such as compilation generate new system components from existing ones. *Construction rules* record how existing components were generated and how future components should be constructed. For example, a typical construction rule might apply the "Mesa compiler of September 3, 1986 2:28 PM, with debugging and without optimization" to a source module.

(4) *Verification rules*. It is generally important that certain constraints be met by a system in order for the system to be considered "valid". For example, one such constraint might be that all interfaces in a system have a corresponding implementation in the system. Another constraint might specify that the sources and binaries in a system version agree, that is, that the binaries were compiled from the versions of the sources found in the system version. Verification rules are used to specify and record structural and organizational constraints on software.

2.2 Templates

Jasmine defines a *module interconnection language* [DK76] used to describe software systems. Descriptions written in the Jasmine description language are called *templates*. Templates have a simple declarative syntax that is used to represent the relations between software components. There are only two kinds of constructs that may be declared in templates: sets and functions.

The template in Figure 1 describes a B-tree package. The package consists of four files: BTree.mesa and BTree.bcd (the source and binary for the BTree interface, respectively), plus BTreeImpl.mesa and BTreeImpl.bcd (the implementation).

The "*BTree*: *TEMPLATE*" construct names the template (and hence the system described) as "*BTree*". The rest of the template consists of set and function declarations. Sets and functions are described in the next paragraphs.

Three special sets are present in most templates: the *COMPONENTS*, *IMPORTS*, and *EXPORTS* sets. The *COMPONENTS* set enumerates the modules and subsystems that make up a system. Each of these components must in turn be described elsewhere by a template with that component's name. For example, the BTree.mesa module might be described by the following template.

```
BTree.mesa: TEMPLATE = BEGIN END.
```

This template is *null*, since no internal structure of BTree.mesa is described. Components with null templates are *atomic components*, and are considered to be indivisible. The choice of the granularity of atomic components is left open. For example, BTree.mesa could alternatively be described as the set of constants, types, and procedures found in the BTree.mesa interface.

```
BTree.mesa: TEMPLATE = BEGIN . . .
    COMPONENTS: SET = {Const1, Const2, . . .,
    . . ., Type1, Type2, . . ., Proc1, Proc2, . . .};
. . . END.
```

Proceeding in the other direction, the BTree package could itself be a subsystem of a larger system.

```
Jasmine: TEMPLATE = BEGIN . . .
    COMPONENTS: SET = {. . ., BTree, . . .};
. . . END.
```

The declaration of components in the *COMPONENTS* sets is used to represent the module decomposition hierarchy of the system, as described in section 2.1. When a system is decomposed into subsystems and modules, the names of the subsystems and modules must appear in the *COMPONENTS* set. A shared subsystem is represented by naming the subsystem as a component in two distinct templates.

The *IMPORTS* set lists components of other subsystems that are referenced in the template. For example, the name Heap/Heap.bcd refers to the Heap.bcd interface in the Heap subsystem. Heap.bcd must be present in the *IMPORTS* set, since it is not a component of *BTree*, but is used in the declaration of *IsCompiledImporting*. The *EXPORTS* set specifies components that can be imported by other subsystems. For example, the name BTree/BTree.bcd can appear in other templates, whereas BTree/BTreeImpl.mesa cannot, since BTreeImpl.mesa does not appear in the *EXPORTS* set. The *IMPORTS* and *EXPORTS* sets ensure that templates are themselves modular.

Collectively, the *COMPONENTS* and *IMPORTS* sets make up the *naming domain* of the template. For a template to be semantically correct, all of the names that appear in the remaining sets and functions must first appear in one of these two sets.

User-declared sets are used to group components that share a common property, as described in item (1) of section 2.1. For example, all of the elements of the user-declared *Sources* set are Mesa source modules. Set expressions containing the union, intersection, and difference operators can be used in the declaration of sets, as illustrated in the declaration of the *Binaries* set.

Functions represent relationships among the elements of a template. For example, the *IsCompiledUsing* function shows the dependency relation between a compiled binary and the files needed to produce it. Functional expressions may appear on the right side of function declarations, as in the definitions of *IsCompiledUsing*, *IsUsedBy*, and *InducesRebuildOf*. Several functional operators are available (e.g., **PLUS**, **INVERSE OF**, and **CLOSURE OF**). The **PLUS** operator merges two functions into one, thus the declaration of

```
BTree: TEMPLATE =
  BEGIN
    COMPONENTS: SET = {BTree.mesa, BTree.bcd, BTreeImpl.mesa, BTreeImpl.bcd};
    IMPORTS: SET = {FileSystem/File.bcd, CommonSoftware/String.bcd, Heap/Heap.bcd};
    EXPORTS: SET = {BTree.bcd};
    Sources: SET = {BTree.mesa, BTreeImpl.mesa};
    Binaries : SET = COMPONENTS - Sources;
    IsCompiledFrom : FUNCTION =
      BEGIN
        BTree.bcd ⇒ {BTree.mesa},
        BTreeImpl.bcd ⇒ {BTreeImpl.mesa}
      END IsCompiledFrom;
    IsCompiledImporting : FUNCTION =
      BEGIN
        BTree.bcd ⇒ {File.bcd},
        BTreeImpl.bcd ⇒ {BTree.bcd, Heap.bcd, String.bcd}
      END IsCompiledImporting;
    IsCompiledUsing : FUNCTION = IsCompiledFrom PLUS IsCompiledImporting;
    IsUsedBy : FUNCTION = INVERSE OF IsCompiledUsing;
    InducesRebuildOf : FUNCTION = CLOSURE OF IsUsedBy;
  END BTree.
```

Figure 1. Template for a B-Tree Subsystem

IsCompiledUsing in Figure 1 is identical to the following one.

```
IsCompiledUsing: FUNCTION =
    BEGIN
        BTree.bcd ⇒ {BTree.mesa, File.bcd},
        BTreeImpl.bcd ⇒ {BTreeImpl.mesa, BTree.bcd,
            Heap.bcd, String.bcd}
    END IsCompiledUsing;
```

INVERSE OF and **CLOSURE OF** are unary functionals that create a new function from an existing one. **INVERSE OF** may be thought of as an operation that reverses the direction of the arcs in the dependency graph described by the argument function. **CLOSURE OF** computes a new function that is the transitive closure of the argument function with respect to the naming domain of the template.

Any grouping of a system's components can be described by a set, and any relation between components can be described by a function. Our example describes only compilation dependencies, but other relationships can be similarly defined. Also, we are able to construct relatively complex sets and functions from simpler ones using set and functional operators. For example, given the simple functions *IsCompiledFrom* and *IsCompiledImporting* (which can be derived mechanically from the software being modelled) we defined a function *IsCompiledUsing* that tells us what components are needed to rebuild a binary file. We then defined another function *InducesRebuildOf* that answers the question *"If I change a particular component, what other components are affected?"*

2.3 Images

Specific versions of software components are represented in Jasmine by *images*. A version of a module is typically an immutable file (sometimes called a *revision* [Ti82]). For example, a version of BTree.mesa might be the file "BTree.mesa created on August 5, 1983 11:01:57 A.M.". A version of a system is a collection of versions of its subsystems and its modules.

Every atomic component (typically, a module) in a system is represented by an *atomic image*. An atomic image binds the template that describes the module to a specific version of the module. For example, an image of BTree.mesa binds the template for BTree.mesa to the file of August 5.

Subsystems are represented by *composite images*. A composite image is constructed by binding each name in the *COMPONENTS* and *IMPORTS* sets to images for versions of these elements. For example, an image of the BTree template binds the names *BTree.mesa*, *BTree.bcd*, *File.bcd*, etc., to images for specific versions of these files. As another example, an image for Jasmine would bind the name *BTree* to a composite image describing a version of the BTree package.

An image has *attributes* associated with it. An attribute is a (*name,value*) pair, for example (*Create date, 5-Aug-83 11:01:57*). Attributes describe properties of the system or module described by the image.

2.4 Families

It is often convenient to refer to all versions of a system component. Typically one does this to select one or more of the images that meet certain criteria, such as being created on a given date, containing a bug fix, or being compiled by a given version of the compiler. In Jasmine, the collection of all images (that is, all versions) of a component is called a *family*. A family contains the version history information described in item (2) of section 2.1. A family can be thought of as a table with a row for each image in the family and columns for the attributes of the images. Images are selected by queries across the table, in analogy to selections in a relational database.

Figure 2 shows a family for BTree.bcd. The family has five versions of BTree.bcd, BTree.bcd!1 through BTree.bcd!5. Each version has its creation date-time, its author, and its "action report number" as attributes.

Histories may be imposed on families. A history is a sequence of images in a family. There may be several histories associated with a given family. A history represents (successive) versions of the component that are available to particular classes

Members of BTree Family:

BTree.bcd ! 1	4-Jan-82 9:15:42	Johnsson	AR 15
BTree.bcd ! 2	14-Feb-83 10:22:06	Lauer	AR 543
BTree.bcd ! 3	5-Aug-83 11:01:57	Karlton	AR 672
BTree.bcd ! 4	22-Sep-85 16:47:13	Davirro	AR 1272
BTree.bcd ! 5	2-Jan-86 22:11:04	Fontes	--

Histories:

Development	Project D	System Test	Private
BTree.bcd ! 4	BTree.bcd ! 4	BTree.bcd ! 4	BTree.bcd ! 5
BTree.bcd ! 3	BTree.bcd ! 3	BTree.bcd ! 2	BTree.bcd ! 3
BTree.bcd ! 2			
BTree.bcd ! 1			

Figure 2. *BTree.bcd* Family, with Development, Project, Test and Private Histories

of customers. For example, an operating system kernel family could have a *development history* for kernel programmers and a *customer history* for groups using the kernel to build applications.

Figure 2 shows four histories for BTree.bcd, the Development, Project D, System Test, and Private histories. As shown, a version of BTree.bcd can belong to several histories.

3. NAME EXPRESSIONS

Templates define hierarchical name spaces based on the modular decomposition of a system. For example, suppose templates exist for Jasmine, BTree, and BTree.mesa, where BTree is a subsystem of Jasmine and BTree.mesa is a module in BTree. Then Jasmine/BTree/BTree.mesa is a *path name* from the root of the system hierarchy (Jasmine) to a leaf module (BTree.mesa). This is entirely analogous to the path names imposed by hierarchical file systems.

More general path names are available in Jasmine than in most file systems. We call these *name expressions*. Here are two examples of name expressions involving sets.

(1) **BTree/*Sources***
(2) **BTree/((Binaries ∩ EXPORTS) ∪ Sources)**

Evaluating expression (1) gives the set {BTree.mesa, BTreeImpl.mesa}, as defined in the BTree template. The evaluation of expression (2) computes a new set {BTree.bcd, BTree.mesa, BTreeImpl.mesa}.

Function applications can also appear in name expressions. The application of a function *F* to an argument x is indicated by the expression *F*[x]. The result of function application is a set of names.

(3) **BTree/*IsCompiledUsing*[BTree.bcd]**
(4) **BTree/(*IsCompiledFrom*
PLUS *IsCompiledImporting*)[BTree.bcd]**

Expression (3) results in the set {BTree.mesa, File.bcd} as declared in the BTree template. Expression (4) resembles a lambda expression. The function to be applied is dynamically computed from the expression (*IsCompiledFrom* PLUS *IsCompiledImporting*).

Functions may be applied to arguments that are sets (in fact, the application F[x] of a function F to a name x is just a shorthand for F[{x}]). Expressions (5) through (7) are equivalent.

(5) **BTree/*IsCompiledUsing*[{BTree.bcd, BTreeImpl.bcd}]**
(6) **BTree/*IsCompiledUsing*[*Binaries*]**
(7) **BTree/*IsCompiledUsing*[*COMPONENTS − Sources*]**

The result of evaluating a function against a set is the union of the results of applying the function to the elements of the set (that is, functions are homomorphisms with respect to set union). For example,

***IsUsedBy*[{BTree.bcd, BTreeImpl.bcd}]**

evaluates to the same result as the expression

IsUsedBy[BTree.bcd] U *IsUsedBy*[BTreeImpl.bcd]

4. CONTEXT-RELATIVE NAME EVALUATION

Name expressions are intended to be used wherever file path names are normally used. For example, in Unix a user can type the command

```
cp BTree/* .
```

which copies all the files in the BTree subdirectory to the current directory (indicated by the period). Our goal is to eventually integrate the Jasmine name space with existing file systems, so that users can also issue commands such as the following one, which fetches all of the files needed to recompile the BTree binaries to the current directory.

```
cp BTree/IsCompiledUsing[Binaries] .
```

Name expressions such as these are evaluated against *contexts*. A context is a search path of images and sub-contexts. A name expression is treated as a pattern to be matched against the name spaces defined by the templates bound to the images in the context. The result of the evaluation of a name expression is a set of images, for example, the images for the files needed to recompile the BTree binaries. Here is an example of a "release" context.

```
10.0 Release: Context = [
    "Image for Mesa 10.0 Release",
    "Image for Pilot 10.0 Release",
    "Image for Services 10.0 Release"]
```

Suppose the expression *Pilot/VM/Swapper.bcd* is evaluated against this context. First, the expression is matched against the template for the Mesa 10.0 release image (this match fails), and then against the template for the Pilot 10.0 release image (which succeeds). The result is an image for a specific version of Swapper.bcd.

Jasmine contexts are similar to distributed file systems (e.g., [W*83, S*85]), in that they allow a user to find and access files that have been distributed on many different network sites. Contexts differ from distributed file systems in the following way. Distributed file systems offer a single global name space, whereas contexts link together many disjoint global name spaces. The disjoint name spaces are those defined by the templates bound to the images in contexts.

5. LOCATING AND TRANSFERRING FILES

When a name expression is evaluated (as described in section 4), the result is a set of images that specify versions of the desired files. It is then necessary to locate these files on network file services.

We have chosen not to embed file location information in images. This decision was based on our experience with the DF software, where we found that embedding file location information in system models seriously hampers the replication of software files. Instead, we name images with network-wide unique identifiers (UIDs). By associating the UID of an image with the file described by the image, we in effect provide a universal name space for files.

Recall that we assume that versions of files are immutable. Xerox file services directly support immutable versions of files; on other systems, such as Unix, version control systems like RCS [Ti82] are needed. When a version of a file is replicated, the (identical) replicas can all be named by the same UID, since the replicas are immutable.

We maintain a simple distributed database (which we call *JFile*) that maps UIDs to file locations. The mapping is one-to-many, that is, one UID can map to the locations of several replicas of the file. In a network there may be several types of file service, each with different file naming conventions. For example, the syntax used to name files on Xerox file services differs from the Unix naming convention. Therefore a JFile entry may contain file locations with different formats.

The JFile entry below shows a UID for BTree.mesa mapped to a location on a Xerox file service (called Orion) and a location on a Unix machine (called Wally). The Xerox location includes a version identifier ("!3"). The Unix location includes an RCS file ("BTree.mesa.v") and RCS version number ("1.3").

$\mathrm{UID}_{\mathrm{BTree.mesa}} \Rightarrow$ <
(Orion:CS:UWash)Jasmine/BTree/BTree.mesa!3,
UW-Wally:/u1/jsm/btree/BTree.mesa.v - (1.3) >

It may be necessary to transfer files between different file systems, for example, from a Unix system to a Xerox workstation. To do so, "plug-in" Fetch and Store procedures are registered with the file transfer mechanism on the workstation. These procedures encapsulate the file system specific information that is required to transfer files between a pair of file system types.

6. JASMINE TOOLS

There are several primitive operations that reference and manipulate Jasmine system models. For example, Jasmine supplies operations for constructing a template, evaluating a name expression relative to a context, and constructing an image with reference to a context and a template. Jasmine tools are built on top of these primitives.

We have designed a suite of tools that support a development methodology currently in use at Xerox. These tools include:

- The *Browser* allows a user to browse system models. The system structure and dependencies can be shown in both textual and graphical representations.
- The *BringOver* tool retrieves the files specified by a name expression to the local workstation. This tool is used by developers to construct and maintain systems, and by clients to retrieve tools produced by a group (for example, the latest released compiler).
- The *StoreBack* tool is used in conjunction with BringOver. Once a developer has built a version of a system, StoreBack is used to define new images with appropriate attributes and to store files onto public file servers.
- The *ReleaseTool* is used to generate system releases. Once a development group has completed the development of a version of a system, the ReleaseTool is used to check the system for consistency, to set attributes, and to distribute files to customers.

7. COMPARISON WITH RELATED SYSTEMS

In this section, we compare Jasmine's main abstractions with those found in other programming-in-the-large systems. The systems used for comparison are Make [Fe79], DSEE [LC84], SML [Sc82], Adele [Es85], and the DF software [Le83].

- *Templates* do not closely resemble any other module interconnection language. Templates allow arbitrary groupings and relationships to be defined via sets and functions, respectively. Functional operators (including functional composition) provide an unusually powerful mechanism for defining and computing complex relationships from existing relations.

  Unlike some other systems (for example, Make and DSEE), templates do not contain explicit construction rules and system constraints (see section 9). Nor do templates contain version or file location information as do SML and the DF software. This allows templates to be used as system structure descriptions with unbound component parameters and construction semantics.
- *Images* are similar to abstractions found in other programming-in-the-large systems; for example, ***bound configuration threads*** in DSEE, ***bound models*** in SML, and released DF's. The notion of a binding between a system model and versions of the components in the model occurs in all of the systems compared.
- *Families* in Jasmine most closely resemble Adele families. SML and DF rely on an underlying versioning file system to group component versions in directories. Make requires a supplementary version control system such as RCS [Ti82].
- *Contexts* are used together with families to provide a version selection mechanism similar

to DSEE *configuration threads* and the selection mechanism of Adele.

- *JFile* provides the functionality of a distributed replicated file system, while relying on existing file services for the actual storage mechanisms. JFile allows the underlying file servers to differ in type. None of the systems compared offer this combination of features.

8. STATUS

An implementation of Jasmine is currently under way. We have chosen a modest set of goals for this first implementation. For example, the JFile implementation is simplified, attributes of images are not fully implemented, and there is only a simple selection mechanism for families. We chose this strategy to obtain early experience with the basic Jasmine mechanisms. By the end of this year (1986), Jasmine will be managing the development of several projects, including Jasmine itself.

Our early experience with the utility of Jasmine has been very encouraging. For example, our microcode development group has never been able to effectively use existing system modelling facilities, since the microcode programming languages they use lack many features (such as enforced consistent compilation [M*79]). We have been able to develop the semantics necessary (as discussed in section 8) to support their development methodology.

9. FUTURE WORK

We are exploring extensions to the Jasmine design presented here:

- *Construction and Consistency Verification.* Currently, the sets and functions in templates have no explicit semantics (except the *COMPONENTS*, *EXPORTS* and *IMPORTS* sets). They have implicit semantics, however. For example, the *IsCompiledFrom* function shown in the *BTree* template implies the dependency between a binary file and its source.

  Construction rules can be represented in Jasmine by attaching explicit semantics to sets and functions. Construction rules specify the operations used to construct versions of a system. For example, when modelling Mesa systems, the *IsCompiledFrom* function can be associated with a construction rule that applies a version of the Mesa compiler (with appropriate switches) to the source file.

  A related problem is *consistency verification*. Consistency verification ensures that the properties and dependencies implied by the system's templates are consistent with the actual system components. (By verification, we do not mean the process of checking that a component meets a functional specification, nor do we mean verification of algorithmic correctness). For example, the *IsCompiledFrom* function can be verified by checking that a binary was compiled from the appropriate version of its source. *Verification rules* are operations associated with sets and functions that verify the system's consistency.

  We are investigating distributed construction and consistency verification.

- *Heterogeneity.* Programming-in-the-large becomes more difficult when forms of heterogeneity are encountered. In a multi-lingual environment, a system may contain modules written in several programming languages, where each language has its own notion of module interconnectivity. In a computer network, files may reside on a variety of file systems, with possibly inconsistent file formats, naming conventions, and access interfaces. Fully general solutions to these problems seem difficult, but experiments with partial solutions are underway.

ACKNOWLEDGMENTS

Brad Davirro, Dan Conde, W.David Elliott, and Joyce Sawai of the Jasmine working group have all made important contributions to the Jasmine design and implementation.

BIBLIOGRAPHY

[Ar86] Armstrong, S. *Factored Courier*. Xerox Corporation internal memorandum, Palo Alto, CA (March 1986).

[C*85] Clemm, G., Heimbigner, D., Osterweil, L., and Williams, L. "KEYSTONE: A Federated Software Environment." *Workshop on Software Engineering Environments for Programming-in-the-large*, Harwichport, MA (June 1985).

[DK76] DeRemer, F., and Kron, H. "Programming-in-the-Large vs. Programming-in-the-Small." *IEEE Transactions on Software Engineering*, volume 2, number 2, (June, 1976).

[Es85] Estublier, J. "A Configuration Manager: The Adele Database of Programs." *Workshop on Software Engineering Environments for Programming-in-the-large*, Harwichport, MA (June 1985).

[Fe79] Feldman, S. "Make - A Program for Maintaining Computer Programs.", *Software Practice and Experience* volume 9 number 4 (April 1979)

[GB80] Goldstein, I., and Bobrow, D. *A Layered Approach to Software Design*. Technical Report CSL-80-5, Xerox PARC, Palo Alto, CA, (December 1980).

[KH83] Kaiser, G., and Habermann, N. "An Environment for System Version Control." *COMPCON Spring 83*, IEEE Computer Society, San Francisco, (February 1983).

[K*85] Kirslis, P., Terwilliger, R., and Campbell, R. "The SAGA Approach to Large Program Development in an Integrated Modular Environment." *Workshop on Software Engineering Environments for Programming-in-the-large*, Harwichport, MA (June 1985).

[La80] Lampson, B. *System Modelling*. Xerox Corporation Internal Memorandum, (May 1980).

[LC84] Leblang, D., and Chase, R. "Computer-Aided Software Engineering in a Distributed Workstation Environment." *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh (April 1984).

[Le83] Lewis, B. "Experience with a System for Controlling Software Versions in a Distributed Environment." *Symposium on Application and Assessment of Automated Tools for Software Development*, San Francisco (November 1983).

[M*79] Mitchell, J., Maybury, W., and Sweet, R. *Mesa Language Manual*. Xerox PARC technical report CSL-79-3 (April 1979).

[Pa72] Parnas, D. "On the Criteria to be Used in Decomposing Systems into Modules", CACM, volume 15, number 12, (December 1972).

[Sc82] Schmidt, E. *Controlling Large Software Development in a Distributed Environment*. PhD Thesis, University of California, Berkeley (December, 1982). Also Xerox PARC technical report CSL-82-7.

[S*85] Schroeder, M., Gifford, D., Needham, R. "A Caching File System for a Programmer's Workstation." *10th Symposium on Operating Systems Principles*, Orcas Island, WA (December 1985).

[Sw85] Sweet, R. "The Mesa Programming Environment". *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, Seattle (June 1985).

[Th82] Thall, R. "Large Scale Software Development with the Ada Language System." *6th International Conference on Software Engineering*, Tokyo (September 1982).

[Ti80] Tichy, W. *Software Development Control Based on System Structure Description*. PhD thesis, Carnegie-Mellon University, (January, 1980).

[Ti82] Tichy, W. "Design, Implementation, and Evaluation of a Revision Control System." *6th International Conference on Software Engineering*, Tokyo (September 1982).

[TL85] Thomas, I., and Loerscher, J.. "MOSAIX: A Version Control and History Management System" *Workshop on Software Engineering Environments for Programming-in-the-large*, Harwichport, MA (June 1985).

[W*83] Walker, B., Popek, G., English, R., Kline, C., and Theil, G. "The LOCUS Distributed Operating System". *9th Symposium on Operating Systems Principles*, Bretton Woods, NH (October 1983).

[Xe81] Xerox Corporation. *Courier: The Remote Procedure Call Mechanism*. Xerox Technical Report, XSIS 038112, Stamford, CT (December 1981).

[Xe85] Xerox Corporation. *Xerox Network Systems Architecture*. Xerox Technical Report, XNSG068504, Palo Alto, CA (April 1985).

[ZW85] Zdonik, S., and Wegner, P. "A Database Approach to Languages, Libraries, and Environments" *Workshop on Software Engineering Environments for Programming-in-the-large*, Harwichport, MA (June 1985).