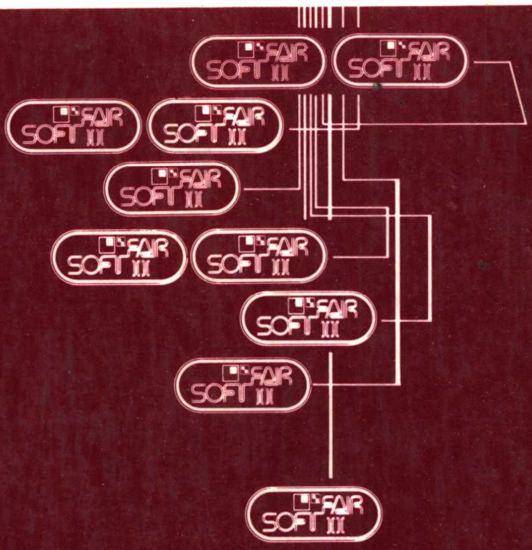
SOFTFAIR II

A Second Conference on Software Development Tools, Techniques, and Alternatives



December 2–5, 1985
San Francisco, California

Co-Sponsored by:

IEEE COMPUTER SOCIETY

Association for Computing Machinery

PROCEEDINGS

0-89791-173-3
CATALOG NO. 85CH2231-9
COMPUTER SOCIETY ORDER NO. 652
ARY OF CONGRESS NO. 85-62329

Association for Computing Machinery

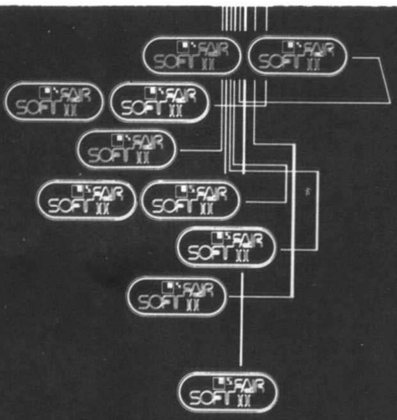
IEEE COMPUTER SOCIETY

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

COMPUTER SOCIETY PRESS

SOFTFAIR II

A Second Conference on Software Development Tools, Techniques, and Alternatives



December 2-5, 1985
San Francisco, California

Co-Sponsored by:

IEEE COMPUTER SOCIETY

Association for Computing Machinery

PROCEEDINGS

ISBN 0-89791-173-3
IEEE CATALOG NO. 85CH2231-9
IEEE COMPUTER SOCIETY ORDER NO. 652
LIBRARY OF CONGRESS NO. 85-62329

Association for Computing Machinery

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

COMPUTER SOCIETY PRESS

ABOUT THIS CONFERENCE

In July of 1983 near Washington, D.C., the Computer Society ran a new form of conference - SOFTFAIR - which combined the traditional paper presentation sessions with indepth tools demonstrations. The goal was to present to the attendees detailed technical knowledge about selected software tools that was not generally available at either the standard technical conferences or the various trade shows. The conference was a big success and the underlying model for the conference has now become somewhat standard for technical conferences.

It is now two and a half years later, and SOFTFAIR II has been organized for the San Francisco area. We believe that this conference both continues the standards set forth in SOFTFAIR I and also shows the advancement that has taken place over the intervening time. We hope that you find the conference and associated tutorials of interest, and we hope that you find these proceedings of value in the years ahead.

Putting together a conference like this requires the help of many individuals. I would like to thank the following people for their help in setting up the technical program:

Program Committee
- Bill Agresti, Computer Sciences Corporation
- Bill Ball, Intermetrics, Inc.
- Ray Houghton, Duke University
- Frank McGarry, NASA Goddard Space Flight Center
- Tony Wasserman, University of California - San Francisco
- Mark Weiser, University of Maryland

Tools Demonstration Coordinator
- Bill Ball, Intermetrics, Inc.

Technical Program Coordinator
- Frances (Andy) Kingery, University of Maryland

I would like to thank all of them, as well as all of the speakers, panelists and session chairs for their contributions.

Marv Zelkowitz
Conference Chairman

TABLE OF CONTENTS

Session 1A—Large Systems Environments

Session Chairman

Mike Evangelist
MCC

Environment Modeling and Activity Management in Genesis

C.V.Ramamoorthy
Vijay Garg
Rajeev Aggarwal

University of California
Berkeley, CA 94720

ABSTRACT

Genesis is a software engineering based programming environment. It provides integrated and intelligent support for the software development during all phases of the software life cycle. It models the environment using Entity Relation Attribute Model, extended with rules. Based on this model, Genesis supports traceability, layering, reusability and other useful software engineering concepts. Genesis uses a knowledge base for expert resources and activities management. Activity Manager, a rule based system, keeps a watch over developers' activities and warns him of possible inconsistencies besides guiding him through the process of software development.

1. Introduction

The software crisis has made researchers aware of the need for a good programming environment. Genesis, a software engineering based programming environment, provides integrated and intelligent support for software development during the entire life cycle. It provides support for requirements, design, coding, testing and maintenance. Our focus is on requirements, design, testing and maintenance rather than coding. Studies have shown that only 10% of the software development time is spent on coding [BOE 81]. Gandalf[HAB 82], Cornell program Synthesizer[TeRe 81], Interlisp[TeMa 81], Smalltalk[GOL 84], Cedar[TEI 84] already provide good support for the development phase. Maintenance is not as well studied. The existing systems for maintenance such as Source Code Control System(SCCS)[ROC 75], Code Control System(CCS)[BAU 78] and Revision Control System(RCS)[TIC 82] provide just version control. Design And Configuration Management system(DACOM)[HAW 83] is more of configuration manager. EPOS[LAU 83] does not support software libraries, metrics or rules. SDMS[SHI 82] is interesting because it is the first configuration management system that supports software evolution using a hierarchical change control structure. None of the above systems, provide integrated support especially with requirements. They do not have any notion of rules and therefore they are more rigid in their structure.

Most environments do not provide a uniform model to the user. However, there are few recent attempts in this direction. Gandalf provides a syntactical tree model uniformly. Smalltalk provides an object oriented environment. ROSI[KOR 85] advocates a relation model as the operating system interface. Recently SAGA[KIR 85] has proposed the use of the entity-relationship like model. Genesis has extended the entity-relationship-attribute model with a set of rules to provide a framework of the environment. This way, the environment has the knowledge of software resources and the development process. This is in contrast to present programming environments that provide hierarchical file system as the framework to store and retrieve resources. Also, they have very little information about the relationship between the resources.

Genesis is aimed at helping in programming-in-the-large. It abstracts out concepts of resources and activities, and provides automatic management of these. Resource Management in Genesis supports reusability, traceability, family concept, layering and complexity metrics [RAM 85a]. In this paper, we focus on the *activity management* in software projects. Activity Management, not to be confused with configuration management, deals with active parts of the project. A project is just a sequence of activities which are defined as, invocations of software tools to manipulate software resources. The aim of the activity manager is to keep a check over this sequence by a rules based protection mechanism, to decrease the length of this sequence by automation of many activities, to warn about potentially inconsistent activities and to guide the developers through this sequence.

Current systems do not put proper emphasis on the integration of requirements with later phases of software development. Genesis attempts to do so by providing traceability between requirements and other software resources. As a part of Genesis, we have transported Requirements Statement Language/ Requirements Engineering and Validation System RSL/REVS[BEL 77] from VAX VMS to UNIX† [RIT 74, KER 81], 4.2 BSD with further enhancements in modularity and understandability[CHEe 85].

VMS is a trademark of Digital Equipment Corporation
† UNIX is a trademark of Bell Laboratories.

© 1985 ACM 0-89791-173-3/85/0012/0002$00.75

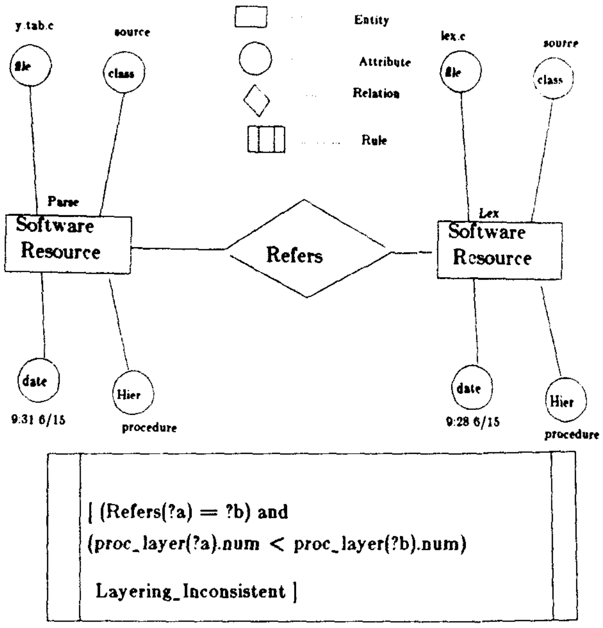
The rest of the paper is organized as follows. Section 2 describes the model that is presented to the user. Section 3 explains the advantages of the model and Software Engineering concepts supported. Section 4 discusses Genesis briefly. Section 5 focuses on Activity Management in Genesis. Section 6 discusses the design of Activity Manager. Section 7 summarizes the status of the project. This is followed by conclusions and appendices.

2. Formal Model

We think that the major weakness of current programming systems is the ineffective hierarchical model of resources, and the lack of emphasis on relations among resources. For example, in most conventional programming environments, users deals with *files* which act as *resources*. These files are independent of each other and are organized in a tree form. This model, we contend, is too weak to support good software engineering principles, because files themselves are discrete entities having no relations at all among them. Furthermore, files or resources have few built-in attributes. Therefore, Conventional Programming Environments are ignorant of contents of files and treat them alike. Thus, retrieving a resource is primitive. For example, UNIX has *ls* and *file* commands to retrieve resources. These primitive commands result in displaying too much redundant data, thus hampering productivity. The software designer has to keep all the bookkeeping himself, and has to make artificial directories to keep resources organized. The designer has to deal with relocation of resources, maintaining backups, and removing resources manually. The problem becomes more acute when he has to deal with a big project using hundreds of files spreading in a maze of directories. Decomposition of a program into smaller modules also result into larger number of files discouraging the user to break his program. Unix alleviates the problem by providing many options with *ls* command but the problem is essentially unsolved. The first task we took, was to provide a more powerful model and therefore furnish more sophisticated traceability and knowledge about resources.

In search of a more powerful model, we abstracted out the concept of objects, links between them and their attributes. With this abstraction, Entities Relations Attribute(ERA) model[CHE 83a] was the closest to our requirements. ERA models information as consisting of *entities*. Entities are similar to nouns in English. They have *attributes* that correspond to adjectives in English. They also have *relations* that correspond to verbs in English. Out of many ERA models proposed[CHE 83b], we are using the model with following restrictions. Only entities have attributes, that is, relations do not have any attribute. We allow only binary relationships between entities, although relationships could be many-to-many. They are directional and relations in one direction are referred as *primary* while in opposite directions as *complementary*. As shown by Chen[CHE 83b], these restrictions do not decrease the expressive power of the model. They make the retrieval efficient and the user model more simple and elegant. This model lets us define relationships between two entities or attributes of some particular entity. However, this model still suffers from a major weakness. It cannot express rules that govern these entities, or actions required in presence of some conditions. To remove this weakness, we have extended the model to incorporate rules in it. A rule is a composition of a condition and an executable entity. Whenever the condition is satisfied, the entity is invoked. The condition is expressed using first order predicate logic involving entities, attributes and relations. The model is illustrated in figure 1.

The figure shows two software resources, *lex* and *parse*, related by a relation *Refers*. These resources have attributes like *time, hierarchy, classification* and so on. The rule states that if any software resource refers another software resource then it should be at a higher layer. *Proc_layer* is a relation (not shown in figure) which links this procedure to the layer, of which it is a part. The syntax of the rule used is explained later.



```
[ (Refers(?a) = ?b) and
(proc_layer(?a).num < proc_layer(?b).num)

Layering_Inconsistent ]
```

fig 1 : Our Model of the Environment

3. Model and Software Engineering Concepts supported

With Entity Relations Attribute Rules (ERAR) as the basic model many of the recognized software engineering features are supported. The importance of the software life cycle is emphasized and an attempt is made to reduce the most time consuming phases. Various entities, relations and attributes are described next. The subsequent discussion also points out important underlying concepts supported.

3.1. Entities

There are three primary entitiy types in an environment. Besides these, developers can add their own entities types to make system more suitable to their project.

3.1.1. Software Resources: These entities correspond to various resources that are used in a software development project. In Conventional Programming Environments all these resources are treated as files and are managed by programmers themselves. Examples of such entities are: requirements specification for a project, design of a layer, design of a module, source of a module, test cases for a module and documentation of a module. Also the user is free to define his own resource type. He can add relations and attributes to those resources to get traceability of the form he desires.

3.1.2. Software Tools: These entities correspond to tools which manipulate software resources. They have certain attributes and relations with other entities. Examples of such tools are editors, compilers and profile generator. These entities will be input to system using Entity Specification Language. As more tools are developed they are added to the database making the system evolvable.

3.1.3. Software Personnel: These entities correspond to the personnel who use *Software Tools* to manipulate *Software Resources*. Modeling of personnel lets Genesis distinguish between their responsibilities and hence enforce protection on other entities.

3.2. Attributes

An attribute describes functional and non-functional aspects of a particular software entity. It consists of two parts - attribute name and attribute value. Most attributes are managed by the system itself. The user can access software resources using queries qualified with these attributes. They are as follows.

3.2.1. Classification: This attribute is valid only for software resources. A Software resource abstracts various kinds of texts used in software life-cycle. The value of this field could be *requirements, design, source, test_case, document, library or object code*. This way each entity gets a label according to its function. Higher layers and rules use this attribute to provide semantic help to developers.

3.2.2. Hierarchy: This attribute classifies each entity into one of the hierarchy as shown in figure 2. Our hierarchy consists of - family, member, layer, module and procedures. With this attribute, we support software family concept and layering.

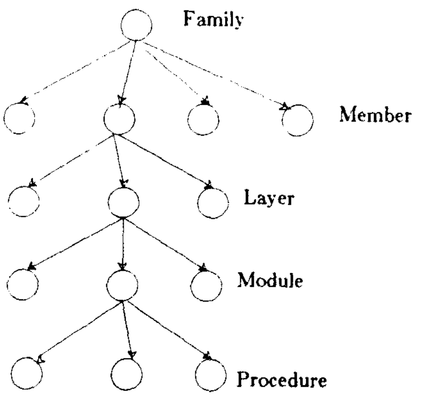


fig 2: Hierarchy classification of resources

Two systems are considered part of the same *Software Family*[PAR 76] if it is easier to understand them by looking at their similarities first and then at their differences rather than individually. *Software Family* concept is important because it captures the notion of *evolution* of a system. Various releases of a system can be considered part of a same family. *Layering* has been accepted as a standard software development technique[DIJ 68, PAR 79]. With the layered structure the development simplifies because of information hiding. It also results in greater modifiability of the system, because a layer can be replaced by another without worrying about other layers. This way new *members* can be generated for a family.

3.2.3. Other Attributes: *Status* describes whether the entity has been completed or not. *Protection* lets higher layers of Genesis provide protection. *Time_Created, Time_Modified* and *Time_accessed* let above layer check consistency between data dependent files. For example, *Make*[FEL 78] can use such attribute. *Version Number* can be used to provide version control of various entities. *Language* and *Preprocessor* can be used to generate makefile and scriptfile. *Locked* can be used for providing resource level synchronization.

Complexity can be used for providing metrics guided design methodology[RAM 84, RAM 85b]. Most of the work in complexity metrics is done at the source level[TSE 83]. Genesis intends to apply metrics at requirements and design level. Further, these metrics can be used to guide the process of designing, a methodology termed Metric Guided Design Methodology in Genesis.

Functional_Keywords are used to retrieve resources from software libraries. Higher layer of Genesis can use this field to increase reusability which itself has been recognized as the most effective method of reducing software life-cycle cost[BIT 85]. Thus, usage of functional_keywords and the abstraction of resources also encourage reusability of other entities besides source code. For example, often when the source code is too machine dependent, requirements and design can be reused. Non-Functional attributes are provided to help retrieval of resources from libraries. Example of these attributes are - *Reliability, Memory_Requirements, Performance*. The current set of attributes provided in Genesis are given in Appendix A.

3.3. Relations

Only binary relations are allowed in Genesis. Relation is the primary mechanism to provide traceability. Traceability among requirements, design and source code is important for both software managers and programmers. It provides means to verify that all requirements are provided and tested. It also helps in software evolution as impact of a *modification request*[ROW 83] can be traced down to modules affected.

3.3.1. <Classification>_to_<Classification> Relation: This set of relations is used for tracing from one class of entities to another. For example, a requirement entity is related to one or more design entities by this relation. With this specification, all designs that correspond to a specific set of requirements can be traced.

3.3.2. <Hierarchy>_to_<Hierarchy> Relation: This set of relations lets Genesis trace from a higher level of hierarchy to a lower level. For example, all modules corresponding to a layer can be traced from the layer specification.

3.3.3. Other Relations: *Refers* is used for tracing resources depending on their reference. *Contains* is used to implement libraries. *Inputs, Outputs* and *Affects* provide other forms of traceability similarly. *Author* and *Owner* can be used to provide access control. A complete list of relations is given in appendix C.

3.4. Rules

Conventional Programming Environments do not have enough information about resources or the software development process. To solve this problem, Genesis has a knowledge base which lets Genesis show an *intelligent* behavior. Rules provide Genesis a framework of detecting conditions in the knowledge base which require some action. Genesis uses *rules* to deduce *intelligent* conclusions which relieves the personnel from these tasks. *Rules* also support *extendibility* and *adaptability*. Genesis can be adapted to any particular environment by adding extra rules. The purpose of these rules are described below:

3.4.1. Methodology Enforcement: The methodology adopted for a particular project can be enforced using rules. For example, a rule which states that a requirement resource should be analyzed before developing design resource, will warn the programmer of his intentional or unintentional oversight.

3.4.2. Checking Inconsistency: With multiple programmers working on hundreds of software entities to develop different sections of a big program, inconsistencies arise easily. Using rules Genesis can detect these inconsistencies and warn the personnel. For example, a rule which states that *if the date_created of entity1* is less than *date_modified* of *entity2* that generates the *entity1* then *entity1* should be generated again, lets us check that entities being used are up-to-date. Activity Manager uses rules extensively to provide an expert assistance to the developer. More examples of such rules are given in the section on Activity Manager.

3.4.3. Automating the tasks of deletion, documentation and testing: Many tasks performed by developers can be automated using rules. For example, a rule which states that a resource not accessed in the past one year should be put on backup, can be be used to cleanup resources automatically. A rule which states that all test cases should be applied again after a change in a resource, can be used to apply existing testcases and get differences with the previous results.

Some other important rules are given in appendix A. Genesis uses Ingres[STO 76] database to implement this knowledge base.

4. Description of Genesis

We have focused our attention on programming-in-the-large, where the number of entities is very large. This lets in problems of resources and activity management, which is handled by Resource Manager and Activity Manager respectively.

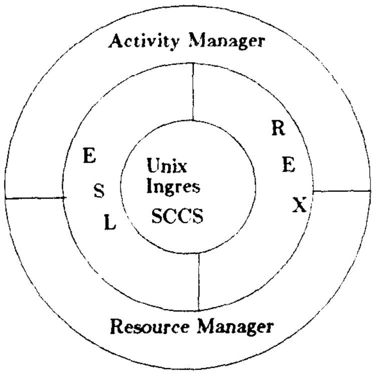


fig 3: Components of Genesis

These and other visible components of Genesis, as given in fig 3 are described next.

4.1. Entity Specification Language System (ESL)

To support Entity Relation Attribute Rules model, we have implemented an Entity Specification Language system that lets software personnel insert, modify or delete various entities in the system. Genesis uses a database to store all this information. An example of interaction with ESL is as follows.

```
Entity John Smith: User
Type: programmer /* an attribute */
Responsible_for: IO_Interface, Error_Reporter /* related to
End
```

4.2. Resource Extractor (REX)

Another subsystem of Genesis, Resource Extractor, lets programmers extract resources with attribute or relation qualification. Using this system, the programmer can trace through various resources. To illustrate the usefulness of the model, we present some example queries of REX. The syntax of a query is as follows:

```
/* Retrieve all those members from set-name2
which satisfy attribute and relation qualifications and put
them in set-name1. */
SET <set-name1> = <set-name2> { <attribute-qualification>
                        | <relation-qualification> }

/* List all elements from set-name which
satisfy attribute and relation qualifications. */
LIST <set-name>{ <attribute-qualification>
          | < relation-qualification> }

/* List all the source code affected by
  a change in file: Getcommand.c */
List Software_Resource affected by Getcommand and
          (classification = source_code)

/* Give me all requirements having high performance
but the corresponding sources having low performances.*/
/* step 1 - get all requirements with high performance */
Set High_Req = Software_Resources
            Classification = Requirements
            Performance = High.

/* step 2 - get corresponding Source_code with low performance */
List Software_Resource with
            Classification = Source_Code
            Relation req_to_source set High_Req
            Performance = Low.

/* Give me all testcases relevant for the source check_for_err */
List Software_Resource with
            Classification = Test_Cases
            Relation source_to_test check_for_err.
```

Of course, the user is not expected to type as much as shown above. Macros and other syntactic conveniences can be provided by above layers. Right now these are missing from the system.

4.3. Resource Manager

Resource manager is responsible for managing above entities. It supports software family concept[PAR 76]. A system is considered as *evolving if it* keeps changing over time. Resource manager supports this notion of *evolution*. The concept of generic and specific information is introduced to deal with common and specific properties of any subsystem. Reusability is supported by providing libraries at various levels - module library, layer library and member library. These libraries contain not only code, but also requirements, design and test cases. Thus the programmer is encouraged to use requirements and test cases, when the code cannot be used because of system dependence. These resources could be retrieved using keywords. Resource Manager also provides version control and let user trace resources using development history. Traceability is provided between members, layers and modules. The prototype of such a system is ready and is in the process of being applied to existing software to gain more experience in this field. The details are in [RAM 85a].

Some future work planned in this area includes metric guided design methodology. Currently most complexity metrics are useful only at the testing and maintenance phase. We intend to apply metrics at the requirements and design level. With these metrics, the resource manager could act as an expert peer suggesting appropriate design out of possible ones, estimating cost and schedule of the project and the impact of a modification request on the system.

4.4. Activity Manager

The goal of Activity Manager is two-fold, to keep a watch over personnel activities and to relieve the developers from menial activities. Thus it acts as programmer's assistant on one hand and methodology enforcer on another. Activity Manager has a database of all software resources, tools that manipulate these software resources, and the project personnel. It also has a knowledge base consisting of various rules about all these kinds of entities. It uses this knowledge base to generate warnings whenever the developer shows some anomalous behavior. It co-ordinates various developers and checks for two persons trying to modify same entity. It warns the software designer, if it detects any inconsistencies between documentation and implementation, requirements and design, implementation and requirements, implementation and test cases. A few examples of inconsistencies are - programmer thinking that he is debugging the latest version of a source code, programmer assuming a behavior about certain module which already has been changed by another software enginneer, the programmer thinking that bug is located in one particular module whereas it could be located in another untested module. Our emphasis was to formulate the rules that programmers follow in their minds. This will leave programmers to do more creative work.

Activity manager generates an inconsistency file which consists of these inconsistencies besides warning user at appropriate times. It also keeps a log of the developer activities and generates logfile and the goalfile. Goalfile keeps track of things to be done by the developer so that the programmer can find out procedures that remain to be designed, tested or documented. It also generates makefiles and scriptfiles to relieve programmer from making them himself.

Activity manager uses rules to enforce protection of resources. Examples of such rules could be, only manager is allowed to change requirements, a programmer cannot change a module if the module has been given status of a tested module, users of the system are not allowed to change any design, document or implementation. The coupling of Activity Manager and other entities in Genesis is shown in figure 4.

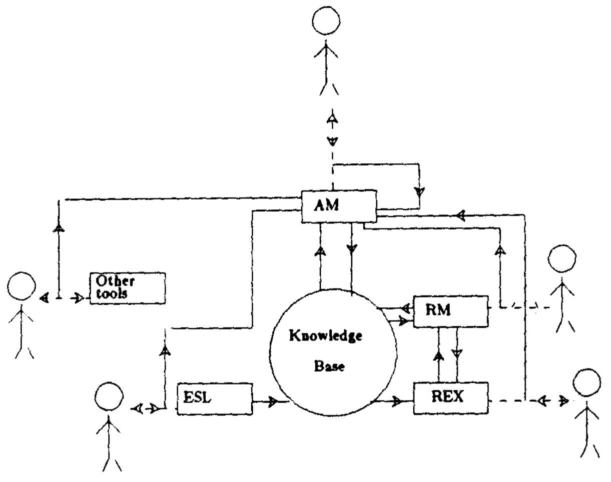


fig 4: Coupling of AM with other components

5. Description of Activity Manager

Activity Manager keeps a watch on the user's activities. In particular, it remembers all commands given by the user. It abstracts out information from these commands to do many bookkeeping jobs for the programmer. AM also has knowledge about the process of software development. It is also aware of the tools used by the programmer. With the help of above, AM assists developers in the following manner.

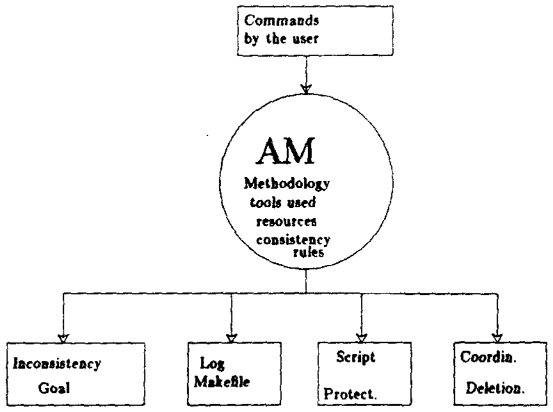


fig 5: Inputs and Outputs of Activity Manager

5.1. Inconsistency_File

This is the most important and difficult task of AM. The programmer generates many inconsistencies while carrying out activities. Examples of such inconsistencies are

- change in requirements, making implementation inconsistent.
- change in some module, making documentation inconsistent.
- moving a module from one layer to another making layering inconsistent.

Thus AM monitors the user's activities and keep a watch over all the potential inconsistencies generated by the user. A typical Inconsistency file would look like as follows:

```
1)Inconsistency
  Change in Requirements
  Entity : Update_Transaction
  Date : May 21,85
  Author : Robert Williams
  Author's comment : The user wanted undo facility
2)Inconsistency
  Change in position of module within layers
  Module : Getindex
  Description : Moved from layer Lex to Parse.
          Used by the lower layer module in Lex :
          check_Reserve
  Date : May 21,85
  Author : John Smith
  Author's comment : null.
```

5.2. Goal_File

Usually a programmer has to design, implement, debug and test many modules/procedures. Further, during the implementation of a procedure in a topdown approach, the programmer uses many unimplemented procedures. Currently, all this is kept track of by the programmer himself. AM alleviates the problem by maintaining a goal file which stores all goals of the programmer. This file is updated by AM automatically as the programmer finishes some part of it or as he assumes existence of some procedures yet to be written. We also have a notion of current goal of the programmer. AM can check that the current goal of a programmer does not conflict with the current goal of another programmer at any given time. That is like *reserve* of the Gandalf project. A typical goal file would look as follows:

```
Readline - design
         - source in C
         - test
         - document
Chkerr   - test
         - document
```

While implementing the procedure Readline, if the programmer uses functions, say getchar or skipblanks, then these too will be added to the list. Activity Manager has, at all times, a list of goals and can remind the programmer of his next goal when he finishes the current goal.

5.3. Log_file

This file is also generated automatically by AM. The aim of the log file is to keep a record of activities performed by software personnel such as, number of man hours spent on the project and breakup of these man hours into coding, testing, documentation. It can be used to evaluate the validity of metrics. For example, reliability models can be evaluated by extracting error history, mean time to fix an error and introduction of errors. A typical Log_file would look as follows:

```
Project Genesis
Programmer Robert Williams started at 9:00 May 15,85
Modules changed
 Get_history : EditorCalls = 6 ; Time = 1:05
 Get_command : EditorCalls = 2 ; Time = 0:32
 Symbol_table : EditorCalls = 5 ; Time = 2:03
Modules Executed
 Get_history : Time = 0:30 ; Test_Coverage = 80%
 Symbol_table : Time = 1:05 ; Test_Coverage = 60%
Modules Documented
 Get_history : Time = 0:35 ;
Finished working on the project at 16:32 May 15,85
```

5.4. Makefile

Currently, makefiles are developed by the programmer manually. In most cases, the data dependency is known through Genesis database. An example of data dependency that could be detected automatically is about *include* files. It is possible to go through these files and generate a dataflow graph which can generate makefile using knowledge about various tools [WAL 84].

5.5. Scriptfile

Activity Manager stores the history of commands issued by the user. This is mainly used to generate default command, default arguments and the script file. This relieves the programmer from making a script file himself when the script file can be deduced from the past sequence of commmands. This is often the case when programmer gives the same series of commands many times, like the conventional edit-compile-load-execute cycle.

5.6. Protection

The protection scheme of the operating system is too naive to handle the protection required in a big project. For example, a programmer should be able to access a particular module only before it is tested. However, if a module is already tested and integrated into the system, a programmer should not be able to change the module without the manager's permission. Thus, the protection mechanism can be extended to suit any project protection requirements.

5.7. Co-ordination among multiple programmers

There are two major functions in this category- first, avoiding read-write and write-write conflict, and second informing the programmer of a possible difference between the behavior of some module as he thinks and as it really is. The difference of the behavior could be because of some change made by another programmer. This relieves the programmer the task of sending mail to everybody who may be using the procedure which has been modified by him. It also relieves programmers of reading through all the mails indicating changes in some procedures most of which they will not be using anyway.

5.8. Warnings about anomalous behavior

In most systems, the user gives commands which are executed by the system directly. But some of these commands, if not used in correct sequence, can result in enormous waste of programmer's time. For example, sometimes the user modifies his source file, forgets to compile it and executes the old version. He may end up spending a lot of time before realizing his mistake. Undo mechanisms are helpless in general when the realization of the mistake itself is costly. AM provides a filter that has knowledge of the expected behavior. This filter generates warnings when the programmer deviates from this behavior. The example above can easily be remedied using this solution.

5.9. Automation of Activities

AM automates many activities. Obsolete information is deleted by incorporating a rule which puts into backup any entity not accessed for some period of time. A part of testing is also automated. Whenever a source module is changed, test cases can be run automatically and the difference with previous outputs noted. By storing extra information, debugging can also be done partially.

6. Design of AM

AM is a rule based system. It uses these rules to enforce protection and check inconsistencies. The nature of the design makes addition of extra rules easy. Thus the system is evolvable. The rules are expressed as logical predicate consisting of relations and attributes of various entities. The following notation is introduced to give an example.

Entity.attribute result is an attribute
relation(entity) result is a set of entities
Rule :: [Condition, entity]
With this notation, example of some rules would be

(1) /* only managers are allowed to change requirements */ [(Affected(?tool).type == requirements) and ((?user).type != manager), Invoke_Modification_Request]

(2) /* if a module has already been certified as tested, it can not be changed by a programmer */ [(Affected(?tool).status == tested) and ((?user).type != manager), Invoke_Modification_Request]

7. Status of Genesis and Future Directions

Genesis is an ongoing project for two years. Requirements system (RSL/REVS) version of unix consists of 20,000 lines of "C" code. ESL and REX reuse most of the software of RSL/REVS. ESE (Resource Manager) runs on Unix 4.2 and consists of about 5,000 lines of "C" code [USU 85]. AM is in the last stages of design [GAR 85]. Meta-rules, which are rules about rules, are being looked into. Ways of classifying rules for performance reasons is another aspect which is being explored. Rules conflict detection and resolution is being worked out. ESL Extension System (ESLXS) is currently being developed which will let user add his own entities, attributes and relations besides those which are predefined. Reverse Engineering (REVENG), in Genesis, refers to algorithms and mechanisms of going backwards from program. The examples are Information Abstraction through Program Slices [WEI 81] and relational view of programs[LIN 84]. This and automatic layering are in research stage. Metrics Guided Methodology (MGM) is considering ways of incorporating metrics in programming environments. The philosophy advocates metrics to serve as guide during the design process. This aspect of Genesis is also in research stage. Large programs are often developed on multiple machines. Providing programmers the facility of Genesis over multiple machines is a nontrivial task. Our long term goal consists of making Genesis cross machine boundaries. The status can be summarized as follows.

Status of Genesis			
Component	Past	Present	Future
RSL/REVS	Implementation	Graphics Interface	DynamicSimulation
ESL	Prototype	Extension with rules	
REX	Prototype	Rules Analysis	
ESE(RM)	Prototype	Experience, Integration	
AM	Design	Prototype	Temporal, Distributed
MGM	Research	Design	Prototype
REVENG	None	Research	Prototype
ESLXS	None	Prototype	

8. Conclusions

Genesis is a software engineering based programming environment. It provides integrated and intelligent support for software development during all phases of the software life cycle. It models the environment using Entity Relation Attribute Model, extended with rules. This model supports traceability, layering, reusability and other useful software engineering concepts. We have integrated requirements with rest of the system. Various forms of traceability between resources are provided. Our system supports reusability through libraries and configuration management through version control. Genesis uses a knowledge base for expert resources and activity management. Activity Manager keeps a watch over developers' activities and warns him of possible inconsistencies besides guiding him through the process of software development. AM generates logfile, goalfile, scriptfile, makefile, and inconsistency-file. It also co-ordinates between multiple programmers and generates warnings for unexpected behavior. It also enforces methodology and the protection on developers. With a knowledge base, it has a wide flexibility of extension through incorporation of additional rules. In short, our system attempts to combine principles of software engineering and artificial intelligence to provide a productive environment.

9. Acknowledgements

We gratefully acknowledge C. Davis, W. M. Evangelist, C. Graff, J. Musa, D. Opferman and R. H. Yacobellis for numerous helpful discussions and comments. We also want to thank our colleagues Y.F. Chen, A. Prakash, W.T. Tsai, and Y. Usuda for helpful suggestions.

10. References

[BAU 78] Bauer, H.A., and Birchall, R.H., Managing Large Scale Software Development with an Automated Change Control System, *COMPSAC* 1978.

[BEL 77] Bell, T.E., Bixler, D.C., and Dyer, M.E., An Extendible Approach to Computer-Aided Software Requirements Engineering, *IEEE Trans. on Software Engineering*, January 1977.

[BIT 85] Bitar, I., Penedo, M.H., and Stuckle, E.D., Lessons Learned In Building the TRW Software Productivity System, *Compcon, Spring 1985*,

[BOE 81] Boehm, B. W., Software Engineering Economics, *Prentice Hall*, 1981.

[CHEe 85] Chee, C.L., A Translator for RSL, *Master's Report*, Electrical Engineering and Computer Science, University of California, Berkeley, May, 1985.

[CHE 83a] Chen, P.S., C.L., ER - A Historical Perspective and Future Directions, *The Entity-Relationship Approach to Software Engineering*, Elsevier Science, 1983.

[CHE 83b] Chen, P.S., C.L., A Preliminary Framework for E-R Models, *The Entity-Relational Approach to Information Modeling and Analysis*, North Holland, 1983.

[DIJ 68] Dijkstra, E.W., THE Multiprogramming System, *Communications of the ACM*, May 1968.

[FEL 78] Feldman, S., Make- A program for Maintaining Computer Programs, *UNIX Programmer's Manual, BSD4.2.*

[GAR 85] Garg, V., A Software Engineering Tools in Genesis, *Master's Report*, Electrical Engineering and Computer Science, University of California, Berkeley, 1985.

[GOL 84] Goldberg, A., Smalltalk-80, The Interactive Programming Environment, Addison-Wesley Publishing Company, 1984.

[HAB 82] Habermann, A.N. et al., The Second Compendium of Gandalf Documentation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, May 1982.

[HAW 83] Hawley, P.J., DACOM: A Design and Configuration Management System, *COMPSAC* 1983.

[KER 81] Kernighan, B.W. and Mashey, J.R., The UNIX Programming Environment, *IEEE Computer*, April 1981.

[KIR 85] Kirslis, P.A., Terwilliger, J.R., and Campbell, R.H.The SAGA Approach to Large Program Development in an Integrated Modular Environment, *Proc. of the GTE Software Engineering Environments for Programming-in-the-Large Workshop, harwichport, MA, June, 1985.*

[KOR 85] Korth, H.F. and Silberschatz, A., A User-Friendly Operating System Interface Based on the Relational Data Model, Internal Report, Department of Computer Science, University of Texasat Austin, Austin, TX, 1985.

[LAU 83] Lauber, R.J. and Lempp, P.R. Integrated Development and Project Support System, *COMPSAC*, 1983.

[LIN 84] Linton, M.A., Implementing Relational Views of Programs, *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.

[PAR 76] Parnas, D.L., On the Design and Development of Program Families, *IEEE Trans. on Software Engineering*, March 1976.

[PAR 79] Parnas, D.L., Designing Software for Ease of Extension and Contraction, *IEEE Trans. on Software Engineering*, March 1979.

[RAM 84] Ramamoorthy, C.V., Prakash, A., Tsai, W.-T. and Usuda, Y., Software Engineering: Problems and Perspectives, *Computer*, October 1984.

[RAM 85a] Ramamoorthy, C.V., Prakash, A., Tsai, W.-T. and Usuda, Y., Genesis: An Integrated Environment for Supporting Development and Evolution of Software, accepted for publication, *COMPSAC* 1985

[RAM 85b] Ramamoorthy, C.V., Tsai, W.-T, Yamaura T., and Bhide, A., A Metrics Guided Methodology, *accepted for publication, COMPSAC, 1985.*

[RIT 74] Ritchie, D.M., and Thompson, K., The UNIX Time-Sharing System, *Communications of the ACM*, July 1974.

[ROC 75] Rochkind, M.J., The Source Code Control System, *IEEE Trans. on Software Engineering*, December 1975.

[ROW 83] Rowland, B.R., Welsch, R.J., Software Development System, *The Bell System Technical Journal*, January 1983.

[SHI 82] Shigo, O. et al., Configuration Control for Evolutionary Software Products, *Proc. of the 6th International Conference on Software Engineering*, September 1982.

[STE 81] Stenning, V., Froggatt, T., Gilbert, R. and Thomas, E., The Ada Environment: A Perspective, *Computer*, June 1981.

[STO 76] Stonebraker, M. et al., The Design and Implementation of INGRES, *ACM Trans. on Database Systems*, September 1976.

[TeMa 81] Teitelman, W. and Masinter, L., The Interlisp Programming Environment, *Computer*, April 1981.

[TeRe 81] Teitelbaum, T. and Reps, T., The Cornell Program Synthesizer: A Syntax- Directed Programming Environment, *Communications of the ACM*, September 1981.

[TEI 84] Teitelman, W., The Cedar Programming Environment: A Midterm Report and Examination *Xerox Corporation, Palo Alto, CA, June 1984.*

[TIC 82] Tichy, W.F., Design, Implementation, and Evaluation of a Revision Control System, *Proc. of the 6th International Conference on Software Engineering*, September 1982.

[TSE 83] Special Selection on Software Metrics, *IEEE Trans. on Software Engineering*, November 1983.

[TSE 84] Special Selection on Software Reusability, *IEEE Trans. on Software Engineering*, September 1984.

[USU 85] Usuda, Y., A The Design and Implementation of Evolution Support Environment, *Master's Report*, Electrical Engineering and Computer Science, University of California, Berkeley, May, 1985.

[WAL 81] Walden, K., Automatic Generation of Make Dependencies, *Software Practice & Experience, June 1984.*

[WEI 81] Weiser, M., Program Slicing, *Proc. 5th International Conference on Software Engineering*, 1981.

Appendix A
Some Rules used by AM

Only manager can change requirements.

Only manager can change a resource after the resource has been classified as tested.

If requirements are changed and design exists then there is a possible inconsistency between requirements and design.

if design is changed and corresponding sources already exist then there is a possible inconsistency between design and the source code.

if source code is changed after the module has been certified as implemented then there could be inconsistency between documentation and the source code.

if a module is changed by someone else, the programmer may be unaware of the change.

if a bug is discovered by a programmer, the developer of the module may be unaware of it and should be sent a mail.

Users of the system are not allowed to make any changes to any part of the system.

If a module refers a module which exists in a higher layer then the layering is inconsistent.

if i want to affect some module which is already being affected by or used by some other person at the same time, then there are chances of inconsistency.

if last time the programmer used some argument to a command, then he is most probably going to use same arguments once again.

if he gave some command after a particular command and if he has given that particular command once again then most probably he is going to give that command again

if source code modified, but not compiled and the old object run then inconsistency

if he is debugging a module which uses some other module which has not been tested properly then he could be looking for the bug at the wrong place.

if he has changed the source code then a bug could have arisen. This could be checked by checking previous test cases and comparing outputs.

if a file has not been touched for last one year, then it could be put in backup.

if during debugging, if program works correctly on one test case, and does not work on the modified version, then we could probably localize the error in the modules that were recently modified.

Appendix B
Examples of Attributes of Software Resources

Attribute Name	Description
Classification	Requirements, Design, Source, Object, Test_Cases, Document, Library
Hier-classification	Family, Member, Layer, Module, Procedure.
Status	Incomplete, Complete
Description	Text
Version No	Number
Protection	Number
Locked	
Time Created	Number
Time Modified	Number
Time Last Accessed	Number
Size	
Functional_Keywords	array of index terms
Body_File	Text
Preprocessor	Lex, Yacc, Equel etc.
Reliability	Low Medium High
Reusability	Low Medium High
Complexity	any metrics
Memory_Requirements	Number

Appendix C
Examples of Relations between Software Resources

Relations	Description
<Class>_to_<Class>	requirements_design, design_Source relation and so on.
<Hier>_to_<Hier>	family_member relation, layer_module relation etc.
Refers	This relation gives dependency between various entities. Other Software Resources
Author	Personnel
Inputs	other software entities
Outputs	other software entities
Affects	other software entities
contains	other software entities /* used for libraries */

An Environment for the Development and Maintenance of Large Software Systems

K. Narayanaswamy and W. Scacchi
Computer Science Dept.
University of Southern California
Los Angeles, CA 90089-0782

Abstract

The maintenance of large software systems is known to be a costly and complicated endeavor. We describe an environment for the development and maintenance of large software systems which offers the potential to aid in the specific activities relating to programming-in-the-large and maintenance-in-the-large. The environment seeks to improve on existing technology for these activities in several ways: a) it is based on a flexible but coherent underlying representation for *families* of software systems, b) it provides the ***upward-compatibility*** mechanism to support incremental alteration of large systems, and c) it seeks to actively support maintenance-related activities by using the existing system descriptions and their inter-relationships to aid in interactive modification of the existing system. Issues relating to the implementation of the prototype environment and its ongoing evaluation are discussed. The environment and the principles which it embodies provide a substantial base for tackling many problems bearing on the design and evolution of large software systems.

1. Introduction

Much of the complexity of the software maintenance effort stems from a seemingly inherent property of software systems called **continuing change** [1]. This terms refers to the phenomenon that so long as a software system is in use, it is always undergoing change on a routine basis. This intrinsic aspect of a software system has several manifestations symptomatic of a more fundamental problem.

1.1. Some Symptoms of Continuing Change

The ongoing alteration of software systems leads to a host of maintenance problems which still remain major intellectual challenges. Several researchers have examined particular facets of these problems:

- Designing modular systems using guidelines such as information hiding [20], Hierarchical Development Methodology [27], abstract machines [21], abstract data types [8], etc. [28].
- Describing the structure of large software systems in terms of module interfaces and inter-dependencies as in the work on Module Interconnection Languages (MILs) [5], [30], [22], [23].
- Tracking the configuration of system module designs and source code [2], [32].
- Minimizing the recompilation, retesting, and reintegration required in incrementally altered systems [7], [24], [33].
- Estimating the possible impact of system alterations using some metrics-based approaches [34].
- Controlling the proliferation of multiple system versions [32], [9].

While one has to bring diverse skills into each of these areas in order to solve the problems peculiar to that area, it is suggestive that they are all exacerbated by continuing change. None of the above problems is critical for a small software system, or even large systems which are developed with the knowledge a-priori that they are never going to evolve. However, given the size of non-trivial software systems, and the virtual certainty of maintenance alterations, such conditions are not likely to arise in production-quality software systems.

1.2. Goals of our Research

While there is no basis to suggest that the solutions to each of the above problems are exactly the same, it is important to realize that there is a sufficient basis to consider them to be related. We believe that the problems listed in the previous section should be addressed within the context of a uniform, **coherent representation for evolving systems** which provides the means to analyze exactly how alterations affect a software system. Such a representation would be able to