



Calhoun: The NPS Institutional Archive
DSpace Repository

Reports and Technical Reports

All Technical Reports Collection

1990

A Graph Model of Software Maintenance

Mostov, I.; Luqi; Hefner, K.

Naval Postgraduate School

K. Hefner, Luqi, and I. Mostov, "A Graph Model of Software Maintenance", Technical Report NPS 52- 90-014, Computer Science Department, Naval Postgraduate School, 1990.
<https://hdl.handle.net/10945/65203>

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

T56

NPS52-90-014

NAVAL POSTGRADUATE SCHOOL

Monterey, California



A GRAPH MODEL OF SOFTWARE MAINTENANCE

I. Mostov

Luqi

K. Hefner

August 1989

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, California 93943-5100

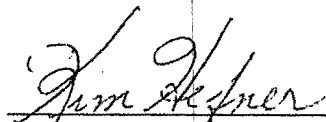
NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

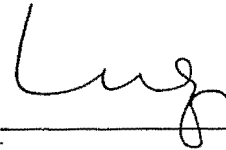
Harrison Shull
Provost

This report was prepared in conjunction with research funded by the National Science Foundation.

Reproduction of all or part of this report is authorized.



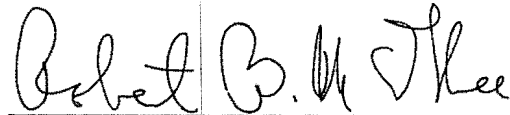
KIM HEFNER
Assistant Professor
of Mathematics



LUQI
Associate Professor
of Computer Science

Reviewed by:

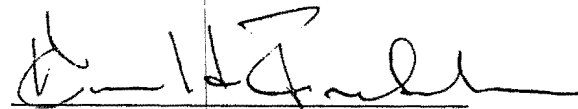
Released by:



ROBERT B. MCGHEE
Chairman
Department of Computer Science



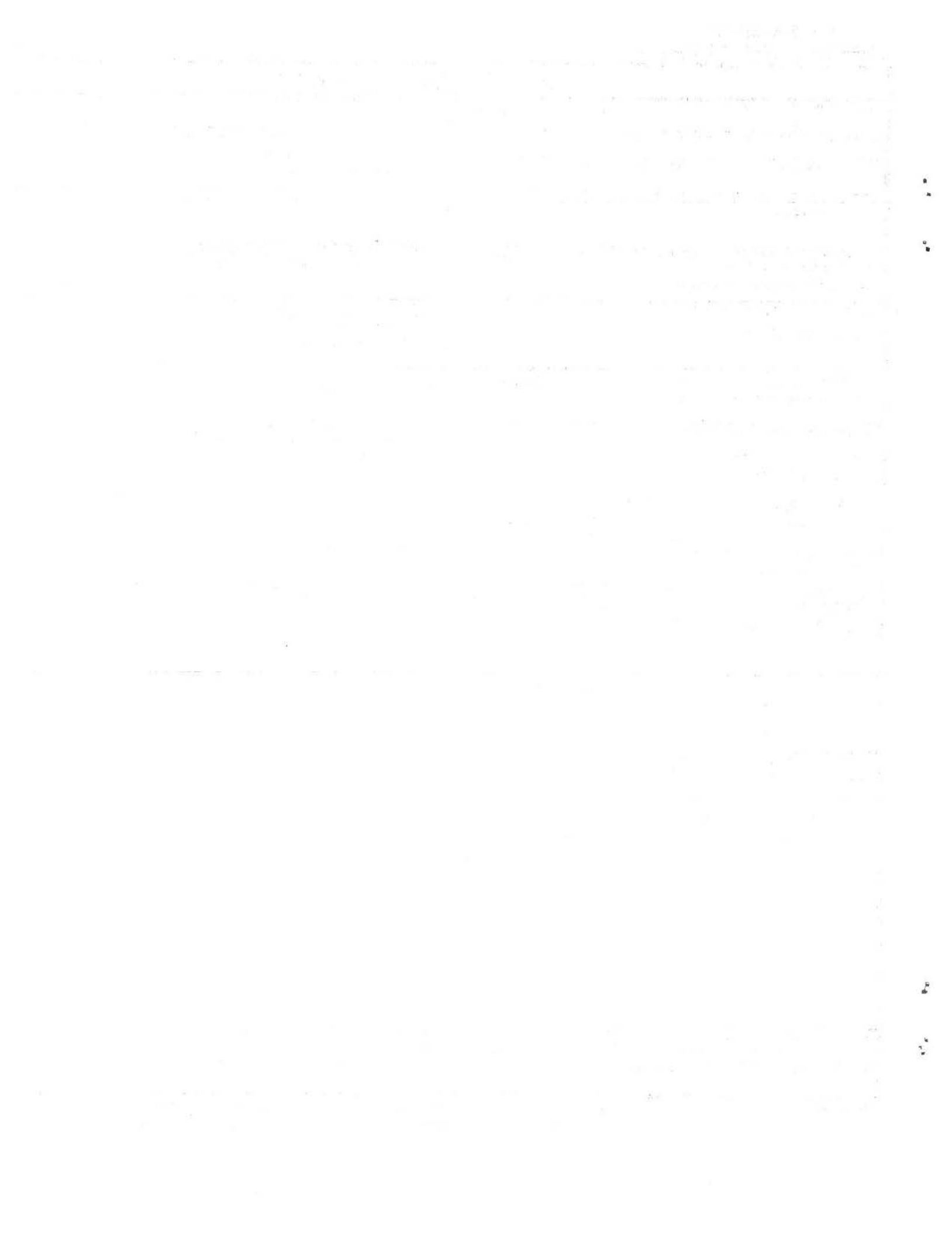
GORDON E. SCHACHER
Dean of Faculty
and Graduate Studies



HAROLD M. FREDRICKSON
Chairman
Department of Mathematics

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-90-014		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5100		7b. ADDRESS (City, State, and ZIP Code) 1800 G Street, NW Washington, DC 20550	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundation	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF CCR-8710737	
8c. ADDRESS (City, State, and ZIP Code) 1800 G Street, NW Washington, DC 20550		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) A GRAPH MODEL OF SOFTWARE MAINTENANCE (U)			
12. PERSONAL AUTHOR(S) I. Mostov, Luqi and K. Hefner			
13a. TYPE OF REPORT Progress	13b. TIME COVERED FROM Mar TO Aug 89	14. DATE OF REPORT (Year, Month, Day) August 1989	15. PAGE COUNT 36
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Effective management of the maintenance process is the most important factor in efficient software maintenance. It requires possession of the updated information about the current state of the maintenance and process control tools that utilize this information. A Model of Software Maintenance based a set of immutable software components and a bipartite graph is described in this paper. This model uses state diagrams to model the temporal behavior of the maintenance tasks, and it incorporates task priorities and precedence, sub-system and baseline definitions, etc.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Luqi		22b. TELEPHONE (Include Area Code) (408) 646-2735	22c. OFFICE SYMBOL 52Lq



A Graph Model of Software Maintenance

**Isaak Mostov
Luqi
Kim Hefner**

**Computer Science Department
Naval Postgraduate School
Monterey, CA 93943**

ABSTRACT

Effective management of the maintenance process is the most important factor in efficient software maintenance. It requires possession of the updated information about the current state of the maintenance and process control tools that utilize this information. A Model of Software Maintenance based a set of immutable software components and a bipartite graph is described in this paper. This model uses state diagrams to model the temporal behavior of the maintenance tasks, and it incorporates task priorities and precedence, sub-system and baseline definitions, etc.

Keywords : Software maintenance, software engineering, graph theory, software model.

1. Relation between Software Maintenance and Configuration

A direct effect of the software maintenance activity is a change in one or more components of the system. These changes affect the configuration of the system, its semantics, and its functionality. Considering the amount and scope of the changes the system is undergoing during its lifetime, complete and effective control over the configuration of the system is imperative.

The relationship between the software configuration of the system and maintenance activities applied to it can be formulated as follows: each maintenance activity is a function on the power set of the system's software configurations; when applied to a subset of a

system's configuration it results in an updated subset of the system's configuration. In a mathematical sense, the system software configuration is generated by the maintenance activities; for any change in the system software configuration there exists some maintenance activity that created it.

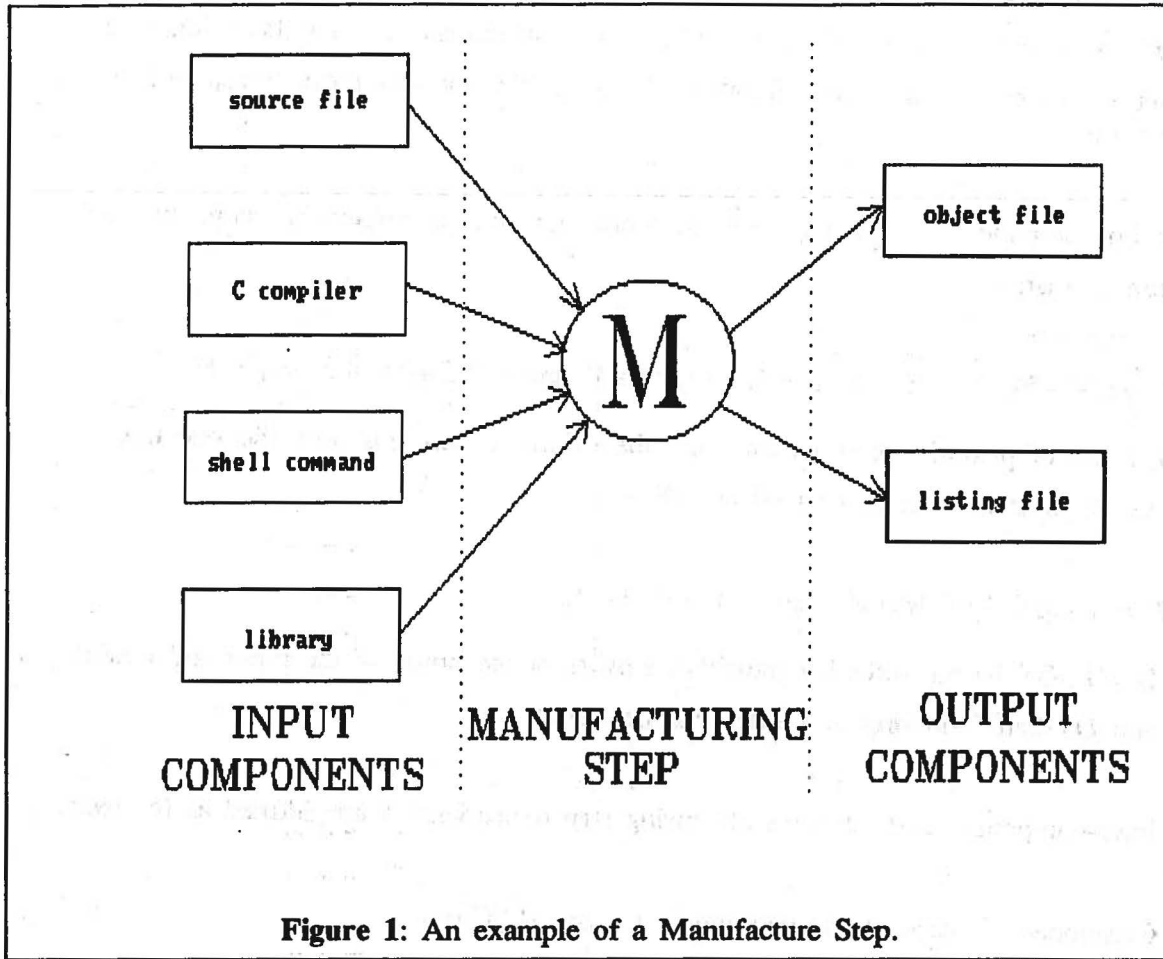
Therefore, we can model the evolution of the system during the maintenance phase of its lifecycle as a graph that consists of software objects that comprise the system configuration, and the maintenance tasks and activities that are applied to these objects. Such a graph captures the semantics of the above principles and allows creation of an abstract mathematical model that incorporates the specifics and necessary information of both software maintenance management and system configuration management and control into one coherent framework.

In the following work we will use the ideas and notations of E. Borison's Model of Software Manufacture [Ref: 1] and of the Graph Transformation Model for Configuration Management Environments [Ref: 2] as the underlying basis for the proposed model. We will refine the original Model of Software Manufacture in order to reflect the specifics of software maintenance and the consistency requirements of the system.

2. Brief summary of the Model of Software Manufacture

The Model of Software Manufacture views software components as immutable objects, i.e., they can be created and destroyed, but once created their values cannot be modified. Any attempt to change such an immutable object creates a new version of this object that differs from the original one. Once created, the software components are not destroyed, they remain alive throughout the lifetime of the whole system and may be used later to spring off new genealogies.

Manufacturing activity (called a *step* in the model) is a derivation relationship between two sets of components: an input set and an output set (see Figure 1). In the original Model of Software Manufacturing the manufacturing step "works" on one or more inputs and "produces" one or more new components. Each invocation of a manufacturing step is considered distinct, whether or not it operates on different inputs.



Both the manufacturing steps and the software components are given unique labels for the lifetime of the system in order to distinguish between them.

The model of Software Manufacture views the system as a finite, labeled, directed acyclic graph (G) of components (C nodes) and manufacturing steps (M nodes). The graph G is bipartite, i.e., manufacturing nodes alternate with component nodes.

The graph G is represented by a tuple $\langle C, M, I, O \rangle$ where C and M are sets of nodes and I and O are sets of edges:

- The set C represents all software components of the system.

- The set **M** represents the manufacturing steps applied to the components of the system.
- The set **I** represents the input relations between components and the manufacturing steps.
- The set **O** represents the output relations between the manufacturing steps and the components.

Since no component can be a product of more than one manufacturing step, the set **O** is restricted so that:

$$(1) \quad \forall M_i, M_j \in M, \text{ If } \exists C \in C \text{ such that } (M_i, C) \in O \text{ and } (M_j, C) \in O, \text{ then } M_i = M_j$$

Also, a set of primitive components (i.e, the primitive configuration, the one that is used to set up the system) can be defined as follows:

$$(2) \quad P = \{ C \in C \mid \neg \exists M \in M \text{ such that } (M, C) \in O \}$$

Let $D^* = (I \cup O)^*$ be the reflexive transitive closure of the union of the input and output relations **I** and **O**, then following properties can be stated:

- The inter-component and inter-manufacturing step dependencies are defined as follows:

$$(3) \quad \text{Component } C_j \text{ depends on component } C_i \iff (C_i, C_j) \in D^*$$

$$(4) \quad \text{Step } M_j \text{ depends on another step } M_i \iff (M_i, M_j) \in D^*$$

- For any component **C** the set of manufacturing steps that are affected by a change in **C** is defined as follows:

$$(5) \quad M_c = \{ M \in M \mid (C, M) \in D^* \}$$

A configuration in the Model of Software Manufacturing is defined as a tuple $\langle G, E, L \rangle$ where **G** is the graph described earlier, $E \subseteq C$ is a set of components designated as exports of the configuration (i.e., components that are designated for a use outside of the configuration), and **L** is a labeling function that distinguishes different components of the system. The graph **G** contains only those manufacturing steps that are necessary to produce an export configuration of the system, i.e., the following holds true:

(6) $\forall M \in M \exists C \in E$ such that $(M, C) \in D^*$

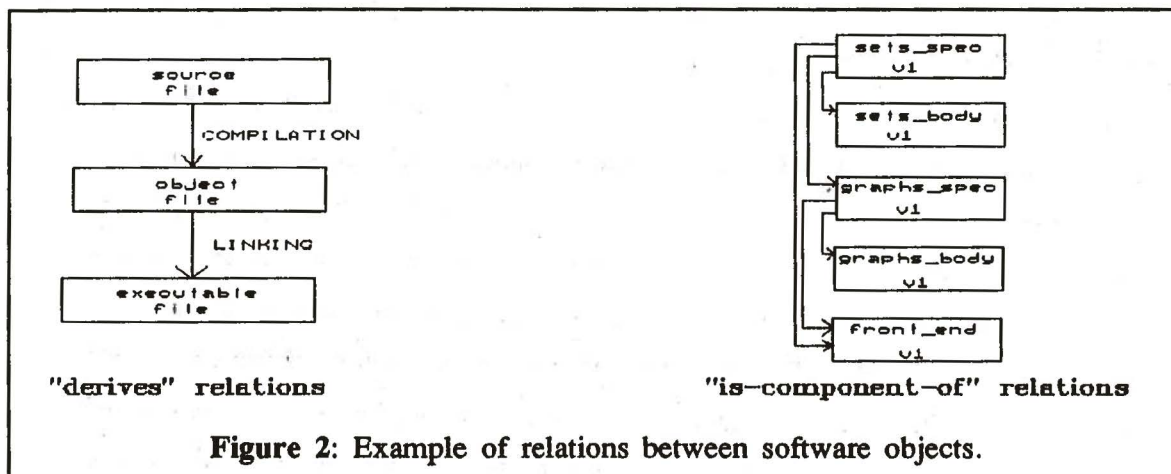
The original Model of Software Manufacture presented briefly above is too general, oriented towards use of tools and application of derivation transformations to some components in order to create others, and is concerned only with manufacturing steps that result in export components. For example, the components in the Model of Software Manufacture are not limited to conventional software modules (e.g., source code files). Actual parameters for tool invocations are considered to be "legal" components of the model. The manufacturing steps have no concrete existence, they are taken to be the derivation relations between inputs and outputs. The Model of Software Manufacture is not suited for the specifics of maintenance tasks, and it must be refined in order to serve as a model of the software maintenance process.

3. Relations between Components

The Graph Transform Model [Ref: 2], classifies software objects into two categories: *re-derivable* and *non-re-derivable*. Re-derivable objects can be automatically reconstructed by applying some tool to some set of software objects. All other objects are considered non-re-derivable (e.g., "source" objects). The software objects may have attributes, which can specify computational procedures that should be applied to the components in order to perform specific transformations.

There exist two important relations between non-re-derivable and re-derivable objects: *is-component-of* and *derives*. These relations have a direction and are easily modelled using digraphs.

The relation "derives" (see Figure 2) is defined between non-re-derivable and re-derivable objects and it represents a transformation of one or more software objects into another (e.g., compilation of source modules into linkable object modules). The "derives" transformations are typed transformations that are applied to objects of a specific type. These transformations are very specific, they are known a-priori, and can be applied automatically using the information about the type of the software object and the attributes of the object itself. The "derive" transformations are associated with the use of software tools in the process



of programming, but they usually are invisible to the management or users of the system.

The "is-component-of" relation (see Figure 2) is defined between non-re-derivable objects only and it represents the use of one component by another component of the system (e.g., use of packages in the ADA programming language). To denote the "is-component-of" relation we will use a convention in which the "is-component-of" relation between components C_i and C_j means that C_i is a component of C_j .

The "is-component-of" relations between components are defined by the system's overall design and module decomposition. These relations may be specified in the component itself (e.g, compiler directives in programming languages - "#include" in C, "with" in ADA, "COPY" in some COBOL dialects) or explicitly stated as attributes representing additional information required for deriving transformations (e.g., library specifications in linking commands). In both cases the relation information is defined a-priori and is stable, relative to the dynamics of the system's changes due to the maintenance process. These stable relationships may change as a result of maintenance, but these changes are few and far in between compared to the changes in the components themselves. Once the "is-component-of" relations are defined for the whole system, such relations can be determined automatically for components using their attributes and the information in the components themselves, and applying knowledge-based techniques.

4. The Model of Software Maintenance

The main objective of the Model of Software Maintenance is to provide a framework that integrates information about software maintenance activities with configuration control. The model is not concerned with the mechanics and the details of the maintenance programmer task and it assumes organizational paradigms that comply with ANSI/IEEE standard on Software Configuration Management [Ref: 3] as follows:

- The management of the software maintenance organization exercises a formal type of change control, i.e., the system configuration changes only as a result of a maintenance action authorized by the management.
- The software configuration management system is used as a tool to coordinate maintenance activities that occur within the context of the system, and the implementation of the control is done utilizing software libraries.
- All of the verified software objects are contained in a controlled software library (i.e. master library) that is under direct control of the maintenance management, i.e., all changes to components of the master library must be authorized.
- The actual programming work is done using the dynamic (programmer's) library which is outside the master library, i.e., when each programmer is assigned to perform a maintenance activity appropriate software objects are copied from the master library to the dynamic one, and the programmer has free access to them; final results of his work are transferred from the dynamic library to the master library when his work has been tested, verified and accepted.
- The products of the configuration (e.g., executable software objects) are derived from the system's configuration repository and installed at the "production" site (i.e., "outside" the configuration repository). These software products are considered to be the "exports" of the configuration.
- Since product derivation may be required at any point of time, the system's configuration must be consistent at all times, i.e., at no time may derivation of executable objects be compromised because of consistency problems of existing completed software objects.

Such organizational paradigms are common to most software development and maintenance organizations that deal with software systems of large and medium size.

a. **Definition of the Model**

The Model of Software Maintenance is comprised of two basic elements: *system components* and *maintenance steps*.

The system components are immutable and non-re-derivable software objects. The system components of the Model of Software Maintenance correspond to the components in the Model of Software Manufacture, with the exception that the components must have concrete existence as software objects of the system. Programming tools and their invocation parameters are not considered to be system components in the model. System components are hence forth called components.

The maintenance steps correspond to manufacturing steps of the Model of Software Manufacture with the following differences:

- A maintenance step is a "representation" of the organizational activity concerned with initiation, analysis and implementation of one request for a change in the system.
- A maintenance step may be atomic (i.e., be responsible for production of at most one output component), or be composed from a number of atomic steps.
- An atomic maintenance step is applied to at most one system component and produces at most one output component.
- A Model of Software Maintenance allows for the existence of empty steps that do not produce output components.
- The model also allows for existence of "dead moves" (i.e., steps that do not lead to production of "useful" components). The existence of such steps is motivated by the need to keep correct records of all maintenance activities, including those that have taken a "wrong turn".
- Deriving transformations are not considered to be maintenance steps and are not represented in the Model.

Additionally in the Model of Software Maintenance, for each maintenance step a scope of a change is defined as all sub-systems (or systems¹) to which the step is applied.

1. The exact definition of a system will be provided later.

The system configuration is an acyclic bipartite graph G of components (C nodes) and maintenance steps (M nodes), in which the components and steps are connected by two relations: inputs (I arcs) and outputs of the maintenance steps (O arcs). The output relations of maintenance steps are defined between a maintenance step and the non-rederivable component it produces. The input relations are defined between a maintenance steps and the system components which are necessary to produce an output component that is consistent with the rest of the system. Naturally, no component that has an output relation with a maintenance step can have an input relation with the same step, i.e., no input and output "feedback" relations are allowed. This extends to any path of relations, thus avoiding cycles and complying with the requirement that G is acyclic.

We will represent the input and output relations by sets of the components for which the relations hold. We will use notation in which for a maintenance step M_q (where q stands for the step's label, e.g., an index in an enumeration) its input and output sets are denoted I_{M_q} and O_{M_q} (respectively). I_{M_q} and O_{M_q} are sets of system components which have input and output relations with the step M_q , respectively.

We can formalize some of the above principle and definitions as follows:

- (7) \forall maintenance steps M , $|I_M| \geq 0$ and $|O_M| \leq 1$.
- (8) M is an empty step iff $O_M = I_M = \emptyset$.
- (9) For all maintenance steps M , if $I_M \neq \emptyset$ then $|O_M| = 1$.
- (10) If component $C \in O_{M_q}$ then $C \notin I_{M_q}$.
- (11) $\forall M_i, M_j \in M$, If $\exists C \in C$ such that $(M_i, C) \in O$ and $(M_j, C) \in O$, then $M_i = M_j$.
- (12) $P = \{ C \in C \mid \neg \exists M \in M \text{ such that } (M, C) \in O \}$
- (13) Let $D^* = (I \cup O)^*$ be the reflexive transitive closure of the union of the input and output relations I and O , then
 - a) Component C_j depends on component $C_i \iff (C_i, C_j) \in D^*$
 - b) Step M_j depends on another step $M_i \iff (M_i, M_j) \in D^*$
 - c) M_C , the set of manufacturing steps that are affected by a change in component C , is defined as $\{ M \in M \mid (C, M) \in D^* \}$

It should be noted, that the properties (11) - (13) of the Model of Software Maintenance are similar to properties (1) - (5) of the original Borison's Model of Software Manufacture, upon which the Model of Software Maintenance is based.

b. Maintenance Step States

During the execution of the maintenance process the maintenance activity, which corresponds to a step in our model, can be in several possible states. These states represent some of the dynamic aspects of the maintenance process that is executed as a result of a user's or maintainer's request.

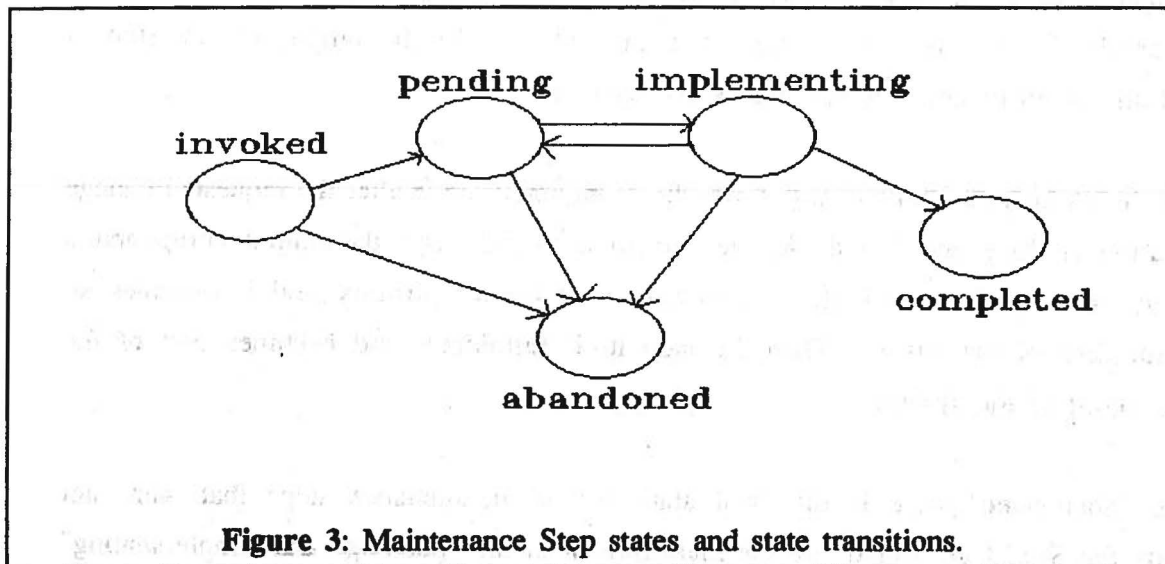
We will define the following five states of a maintenance step:

- Invoked
- Pending
- Implementing
- Completed
- Abandoned

Each of the above states corresponds to several phases of the maintenance process as they are defined in [Ref: 4], and corresponding sub-states can be defined for each of the above states in the implementation of the model.

Transition of a maintenance step from one state to another is performed as a result of an explicit decision made by maintenance organization management. By controlling the states of the maintenance steps, the maintenance management exercises direct control over both the software maintenance process and the system configuration.

For brevity we call a maintenance step by the name of the state it is in, e.g., "pending step", "implementing step", etc.



In the "invoked" state the maintenance step is created according to a requirement for a change in the system. In this state the originated change undergoes analysis which estimates the resources required for the step's implementation, designates inputs and defines the scope of the step. At this stage the maintenance activity is not yet approved for implementation. If the requested change does not become approved, then the maintenance step state becomes "abandoned". In the "invoked" state, the maintenance step is not yet linked to any component of the system.

In the "pending" state the maintenance step is approved but it has not yet started its implementation phase. In order to be implemented, the pending maintenance step must be scheduled and assigned to a programmer. Scheduling of a maintenance step should resolve all possible inconsistencies that may arise as a result of concurrent implementation of the pending steps. A maintenance step in the "pending" state can be "abandoned" (i.e., transferred to "abandoned" state) and forgotten there.

In the "implementing" state the actual implementation and testing of the requested change is performed. At the transition to this state from the "pending" state, the binding of input components occurs, and the output component is created. At this stage, the output component is a placeholder for the future component contents of the maintenance step results, i.e., the output component is "empty" at the beginning, and its contents are produced during

the implementing phase of the maintenance step. The implementing state can be "rolled back" into the "pending" or "abandoned" state, and in such a case the output of the step is invalidated and is no longer a component of the system.

The "implementing" state changes into the "completed" state after the requested change implementation is done and tested. At the transition to this state the output component's contents are frozen and entered into the system component repository and it becomes an approved member of the system. Then the step itself terminates and becomes part of the permanent record of the system.

The "abandoned" state is the final state for all maintenance steps that were not approved by the SCCB or "killed" by the management in the "pending" and "implementing" states.

It should be noted (see Figure 3) that only the maintenance steps in the "implementing" state can be "rolled back" into "pending" state. By doing so all the work that had been performed to implement this maintenance task may be lost due to later changes in the system that may affect the "rolled back" step. Therefore such decisions should be made with insight and great care.

c. Inputs of the Maintenance Steps

The nature of an atomic maintenance step is to incorporate a single change in a single component of the system. We will call such a component a *primary input* of the atomic maintenance step, and we will limit the number of primary inputs for an atomic maintenance step so, that:

$$(14) \quad \forall \text{ maintenance step } M \exists \text{ at most one primary input } C \in I_M.$$

In order to capture the semantics of dependencies of some system components on other components (as defined in (13)), we will introduce the notion of *non-primary inputs* of a

maintenance step. The non-primary inputs of the maintenance step M belong to I_M , the input set of this step, and they are defined by its result:

- (15) For a maintenance step M_q the set I_{M_q} consists of the primary input and all other components that are used in creation of the component $C \in O_{M_q}$ and in the deriving transformations that produce other software objects from it.

It immediately follows from (15) that:

- (16) If \exists an "is-component-of" relation between C_i and $C_j \in O_{M_q}$ then $C_i \in I_{M_q}$.

The non-primary inputs may be determined manually or automatically using knowledge based techniques. If the primary input C_i of a maintenance step M_q is a primitive component (i.e., $C_i \in P$ as defined in (12)), then the non-primary inputs of M_q can be computed from the "is-component-of" relations of system components with the component C_i . Otherwise the inputs of the previous step M_p with $C_i \in O_{M_p}$ may be used to compute the set $I \in M_q$. Facilities for updating the input set of a maintenance step in the pending and implementing states must be provided in the implementation of the Model.

It should be noted that the existence of the inputs for the maintenance step is not a necessary condition (see (7)). There may exist a maintenance step without inputs that produces an output component. Also, there may exist a maintenance step with non-primary inputs only that produces an output component. An example for a such case is the creation of a new component by merging the contents of a number of existing components².

d. Descendence relation, genealogy trees, systems and baselines

We will define a descendence relation by saying that primary input C_i of a maintenance step M is an *ancestor* of the resulting component $C_j \in O_M$, and that the component $C_j \in O_M$ is a *descendent* of C_i .

-
2. It should be noted that there exists another way to represent such merging: one component can be the primary input of the maintenance step that performs the merging, while other components that are used in the merging process, become the non-primary inputs of the merging step.

The descendance is a transitive relation, i.e.,:

- (17) Component $C_k \in O_{M_p}$ is a descendent of C_i iff C_i is the primary input of a step M_p or the primary input of M_p is a descendent of C_i .

In order to distinguish between types of descendance relations for later use, we will call the non-recursive descendance a direct descendance.

The descendance relation defines *evolution genealogy* (called genealogy for shorthand) subgraphs of the configuration graph G . These subgraphs are created using only the primary input and output relations of the maintenance steps. Because the graph G is acyclic, and due to the restriction of at most one primary input for a maintenance step, each element of the genealogy subgraph is a tree. Thus, the following hold true:

- (18) All descendants of a component C belong to the genealogy tree T that has C as its root or one of its nodes.
- (19) There exists a unique path between component C and any of its descendants.

We call a genealogy tree with a component C as its root a C genealogy tree and we will denote it as T_c . A genealogy tree can be a subtree of other genealogy tree(s), or be a spanning tree, which is defined as follows:

- (20) T_i is a spanning tree $\Leftrightarrow \neg \exists T_j, i \neq j$ such that $T_i \subset T_j$.

Using the notion of genealogy trees we can define software *systems* (or products), and we will say that a software system S is uniquely identified by a set of genealogy trees that comprise it, i.e., $S = \{T_i\}$. It should be noted that the whole system is defined by a set of all spanning genealogy trees, and in the case of a sub-system, the genealogy trees T_i are not necessary spanning. Since sub-systems themselves can be viewed as systems, we will not distinguish between them unless it is required. The following rules apply to both systems and sub-systems alike, since software products can be viewed as either systems or sub-systems.

We will say that software system S is *complete* if the following holds true:

(21) $\forall T_i \in S$, if $\exists T_j \mid C_k \in T_i$ depends on $C_n \in T_j$ or C_n is a component of C_k , then $T_j \in S$.

We can express the relations between components, systems and scope of a maintenance step as follows:

(22) Component $C_i \in S \iff \exists T_k \mid C_i \in T_k$ and $T_k \in S$.

(23) Let K be a scope of step M_q with primary input C_i , then $K \subseteq \cup S_k \mid C_i \in S_k$.

The inequality of K and $\cup S_k$ in the above rule means that there may be a situation in which by a virtue of a maintenance management decision an influence of a maintenance step will be limited to particular (sub)systems.

So far we have defined a software system invariably by evolution paths. In order to define the temporal aspects of the system evolution, we will define its *baseline*. A baseline B of a system S (denoted as B^S) is a set of unique representatives of the genealogy trees that comprise the system S , i.e.,:

(24) $B^S = \{C_i \mid (C_i \in T_k \text{ where } T_k \in S) \text{ and } (\forall C_i, C_j \in B^S \ C_i, C_j \in T_k \iff C_i = C_j)\}$.

A baseline of a system must be complete and consistent, i.e., the following must hold true:

(25) $\forall C_i \in B^S$, if $C_i \in P$, then $\forall C_j$ such that C_j is a component of C_i , $C_j \in B^S$. Otherwise $\exists M_q$ such that $C_i \in O_{M_q}$, and $\forall C_j$ non-primary inputs of M_q , $C_j \in B^S$.

A baseline is unique in a context of a system, i.e.,:

(26) $\forall B_k^S, B_n^S$ of a system S , if $\forall C_i \in B_k^S \ \exists C_j \in B_n^S$ such that $C_i = C_j$ and $|B_k^S| = |B_n^S|$, then $B_k^S = B_n^S$.

Evolution of the system can be viewed as an ordered set of its baselines, where the ordering is based upon time of a managerial decision to introduce (or roll-back to) a baseline. It should be noted that basing the ordering on the time of creation of a baseline is not enough,

since it does not take into account the possibility of a roll-backs to a previously defined baseline.

e. **Designating primary input for a Maintenance Step**

Designating a component C as an input to a maintenance step means that this step will take the component C or one of its descendants as a primary input. Such a designation is possible only for pending maintenance steps, and it does not prevent the use of this component by other maintenance steps, i.e., it does not "lock" the input component.

The actual binding of primary input with the designated component can be performed only if the input component is *available* and it is done at the transition time of the maintenance step from the "pending" state to the "implementing" state, after the step is scheduled and assigned to a programmer. After the primary input is binded to the implementing maintenance step, its non-primary inputs are determined.

The primary input component is available for a maintenance step if it is a primitive component or it was created by some completed maintenance step, i.e., it exists and is not being work upon now:

(27) A component C is available iff $C \in P$ or \exists completed M such that $C \in O_M$.

Components that are currently under work by some uncompleted maintenance step may be used as non-primary inputs of a step, and they are considered available the moment they are created by an implementing maintenance step.

The designation of a primary input component to a maintenance step can be *specific* or *generic*. When specific designation is done, the maintenance step will be applied to some specific component that already exists in the system, e.g., to a specific version of the primitive component. There is no possibility of specifically assigning a component that is not produced yet.

In the case of a generic designation of component C, the maintenance step will be applied to the available descendent of C. As mentioned earlier, the actual binding of the primary input to a component that belongs to the genealogy tree of C will take place when the maintenance step begins its implementation phase. For example, in the case of a generic designation, if there are several maintenance steps with the same designated input component C, then the first maintenance step is applied to the component C, the second is applied to the descendent of C, and so on.

It should be noted that, because of the possibility of creating "parallel" genealogies that originate from the component C, the generic designation may not be unique.

f. **Decomposition of Maintenance Tasks**

Sometimes, after analysis of a requested change that initiates a maintenance step, it becomes apparent that the implementation of the original change leads to changes in several system components. The original request for a change is represented as a maintenance step. However, because a maintenance step can be applied to at most one primary input and produces at most one output component, the maintenance step invoked by the original request for change becomes a composite maintenance step; it "spawns" a number of new atomic maintenance steps. In such cases, it is important to record a relation between the composite step and the steps that are "spawned" by it. We will call this process "*step decomposition*", the relation "*spawned*", the composite step a "*spawning step*" and the newly invoked steps "*spawned*". The step decomposition process is recursive, i.e., spawned steps may themselves be composite maintenance steps.

In order to eliminate the possibility of unnecessary relations between composite steps and their spawned steps, the composite maintenance step may not produce any new component by itself, i.e., it is an empty step.

The composite step decomposition takes place when the step's implementation is authorized and it creates a set of spawned steps. These spawned maintenance steps are created directly in the "pending" state, i.e., they are considered to be authorized by the virtue of

authorizing the composite step. The spawned steps behave normally, meaning that they do not differ from non-spawned maintenance steps and they may have relations with one another or some other maintenance steps. Because the spawning step is an empty step, there are no dependency relations between the spawned steps and their spawning steps.

In order to provide consistency in treating composite and atomic steps, the following constraints are imposed on some state transitions of the spawning and the spawned steps:

- (28) The spawning step is transformed automatically from pending to implementing states when one of its spawned steps performs this transition.
- (29) The spawning step performs an automatic transition from implementing to completed state when all of its non-abandoned spawned steps have done so.
- (30) Abandoning a spawning step will automatically abandon all of its spawned steps.
- (31) The transition of a spawning step to abandoned state is done automatically when all of its spawned steps are abandoned.

It should be noted that, since all spawned steps are intended to implement a specific maintenance task, a single maintenance step (atomic or composed alike) can be spawned directly by only one composite maintenance step, i.e., the graph formed by a spawning relationship is a tree.

g. Induced Maintenance Steps

An engineering change in a key component of a software system may compromise the consistency of a systems that belong to the scope of the step by affecting other components of these systems in such way, that some action is required in order to keep them consistent. For example: a change in the specification of some ADA package requires some action to be performed on all other components that use this specific package before any new software product can be successfully derived.

We will define an *induced maintenance step* as a step that must be performed in order to keep the system's consistency due to a result of another maintenance step. The importance

of induced maintenance steps is in alerting the maintainers and the management to changes in key modules of the software product and enforcing constraints on performing any uncoordinated maintenance step on the affected components.

It should be noted that a change in one component may trigger a change in another, which may itself trigger a change in third component, and so on, i.e., the changes may be triggered recursively. We will call a component that originated the change propagation a *triggering* component, and the step that uses it as its primary input an *inducing step*.

Additionally, the propagation of the changes triggered by an inducing step must be restricted to the scope of the inducing step.

In order to define specifically the relevant maintenance steps that are affected by a change in component C , we will introduce the concepts of *latest descendent* and *latest maintenance step*. The latest descendent of a primitive component is a component that is not used as primary input by any maintenance step, i.e.,:

- (32) C_i is a latest descendent of C_j iff C_i is a descendent of C_j and $\neg \exists M_q$ such that C_i is the primary input of M_q .

The latest maintenance step is defined as follows:

- (33) Step M is the latest step with respect to C_j iff the component $C_i \in O_M$ is latest descendent of C_j .

We will refine now the original definition of the set M_C (see (13)) using the notion of the latest maintenance step as follows:

- (34) M_C , the set of maintenance steps affected by a change in triggering component C , consists of latest steps M which have C as their non-primary input and O_M belongs to the scope of an inducing step that implements the change in C .

Since a change in one system component may lead to the inconsistencies with the primitive components (i.e the components that were not produced by any maintenance step), the above definition is not sufficient and we will introduce the notion of an *affected*

component. A system component, whose consistency with the rest of the system is affected by a change in some other component, is called an affected component and the following holds true:

- (35) A component C_i is affected by a change in C_j iff C_i belongs to a scope of an inducing step that implements the change in C_j and either $C_i \in O_M$ where step $M \in M_{C_j}$ or both $C_i \in P$ and $C_j \in P$ and C_j is a component of C_i .

Analogous to the definition of the set M_C (see (34) above), we can define a set of all components that are affected by a change in a component C_j (noted as C_{C_j}), using the recursive nature of the propagation of a change, as follows:

- (36) A component C belongs to the set C_{C_j} if C is affected by a change in C_j or C is affected by a change in $C_i \in C_{C_j}$.

We will say that if $C_j \in I_{M_p}$ is a primary input and the set C_{C_j} is not empty, then the inducing step M_p induces maintenance steps $M_{p(n)}$ ³, where $n=1,2,\dots,|C_{C_j}|$. An induced maintenance step $M_{p(n)}$ takes an affected component $C \in C_{C_j}$ (as defined in (36) above) as its primary input, and produces as its output a new component which is consistent with the direct descendent of the component C_j .

Uncoordinated propagation of the induced steps may cause the transient state of the systems configuration to be inconsistent. For example, a change of the package specifications without coordinated change of the package body may cause some incompatibilities and compromises the consistency of the whole system.

In order to keep the system configuration consistent, the inducing maintenance step together with its induced steps are performed as an atomic step, i.e., an inducing step and all of its induced steps (including those that were created recursively) should appear to perform their transitions from "pending" to "implementing" states and from "implementing" to "completed" states simultaneously. The following rules impose a semi-atomic behavior of inducing/induced steps by introducing the necessary constraints:

3. The notation $M_{p(n)}$ is used as a labeling suggestion for induced steps.

- (37) An inducing step M_q with primary input C_j can start its implementation phase iff all steps $M_p \in M_{C_j}$ are completed.
- (38) An induced step $M_{q(n)}$ with primary input $C_i \in C_{C_j}$ can start its implementation phase iff the inducing step M_q with primary input C_j has already done so.
- (39) An inducing step M_q with primary input C_j can become completed iff all induced steps $M_{q(n)}$ with primary input $C_i \in C_{C_j}$ are completed.
- (40) Any "roll back" transition of the inducing step causes the same transition to be performed on all its induced steps.
- (41) An induced step can be "rolled back" only by "rolling back" all of its inducing steps.
- (42) Abandoning an inducing maintenance step causes all of its induced steps to be abandoned.
- (43) An induced step can be abandoned only by abandoning its inducing step.

The meaning of rule (37) is that the inducing step cannot begin its implementation before it assures that all steps that are affected by it are completed. This means that the induced steps may begin their implementation phase since their primary inputs will be available (see (27)). The rules (38) and (39) mean that the induced maintenance steps cannot begin their implementation before their inducing maintenance step, and the inducing step cannot complete the implementation phase before its induced steps. Other rules mean that the induced step has no reason to exist by itself, without its inducing step.

Because of the dynamic nature of the system's configuration, the influence of an inducing maintenance step M_q with primary input C_j (i.e., the contents of the set C_{C_j}) may vary. The contents of the set C_{C_j} become static as a result of scheduling of the step M_q . Since the induced steps $M_{q(n)}$ depend directly upon the contents of C_{C_j} , they are invoked immediately after the inducing step M_q is scheduled for execution.

Figure 4 presents an example of maintenance steps applied to a system built from five components: `sets_spec`, `sets_body`, `graph_spec`, `graph_body`, and `front_end`. The system represents graphs using sets, performs operations on them and presents the results to the user

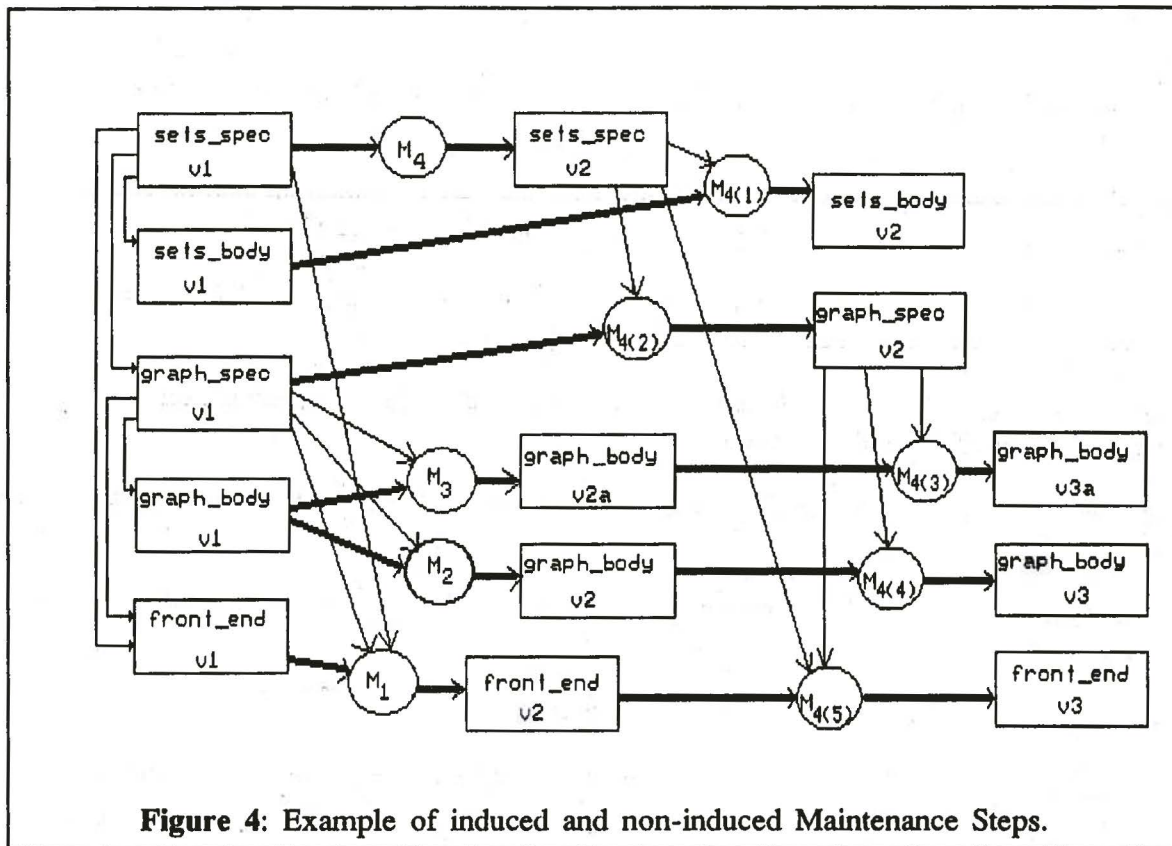


Figure 4: Example of induced and non-induced Maintenance Steps.

through procedures in the **front_end** module.

The example shows an influence of a simple maintenance step on the system configuration, a creation of distinct evolution genealogies, and the creation and propagation of the induced maintenance steps.

The set of primitive components of the system consists of the following five modules:

P = {sets_spec.V1, sets_body.V1, graph_spec.V1, graph_body.V1, front_end.V1}

The "is-component-of" dependencies of the primitive components are defined and shown on the left side of the primitive components in the figure.

In the example, a request for a change in one of the **front_end** procedures initiates a maintenance step M_1 , with the designated primary input **front_end.V1**. After the step is scheduled and implemented, it creates the new component **front_end.V2**, which is a direct descendent of the **front_end.V1** component. The non-primary inputs of the step M_1 are determined using "is-component-of" relation and they consist of components **graph_spec.V1** and **sets_spec.V1**.

A request for a change in the component **graph_body.V1** initiates maintenance step M_2 , which results in the production of a new component **graph_body.V2**. Another request for a change that corresponds to the maintenance step M_3 , has a component **graph_body.V1** as its specific designated input and it results in a component **graph_body.V2a**. Thus giving rise to an evolution genealogy which is "parallel" to the one created by the maintenance step M_2 .

The component **sets_spec.V1** is used as a non-primary input in a number of maintenance steps, and a change in it will affect the following set of components:

$$C_{sets_spec.V1} = \{sets_body.V1, graph_spec.V1, graph_body.V2, graph_body.V2a, front_end.V2\}$$

This set is computed recursively, starting with the set of components that are directly affected by **sets_spec.V1** (i.e., the set $\{sets_body.V1, graph_spec.V1, front_end.V2\}$), and then for each component in the set adding the components which it affects.

A maintenance step M_4 that implements a change in the **sets_spec.V1** component induces the following maintenance steps:

- Step $M_{4(1)}$ that produces component **sets_body.V2**
- Step $M_{4(2)}$ that produces component **graph_spec.V2**
- Step $M_{4(3)}$ that produces component **graph_body.V3a**
- Step $M_{4(4)}$ that produces component **graph_body.V3**
- Step $M_{4(5)}$ that produces component **front_end.V3**

The step M_4 is completed only after all the maintenance steps it had induced (i.e., $M_{4(1)}..M_{4(3)}$) are completed. Thus, the implementation of the inducing and induced maintenance steps is performed atomically, keeping the whole system consistent, i.e., any software object derived from the system components will be consistent with the latest changes incorporated in the system.

h. Priority and Precedence of Maintenance Steps

During the lifecycle of the software system, constraints that reflect the urgency and the partial ordering of the maintenance tasks arise from "real life" situations. These constraints influence the process of software maintenance, and they must be represented in the Model of Software Maintenance in order for the latter to be a realistic model.

We will represent the urgency of the maintenance tasks by assigning a small positive integer value as a *priority* value to each maintenance step that is needed to implement the task. The priority values of the maintenance step represent the relative urgency of the maintenance tasks, and they suggest an implementation ordering of the maintenance steps.

The priorities are assigned manually to the maintenance steps by the Software Configuration Control Board⁴, and they may be changed during the maintenance process (by the SCCB or other appropriate forum) according to the state of the system maintenance and external constraints. The process of assigning priority values to maintenance tasks may use different methods and algorithms and is external to the Model of Software Maintenance itself.

Since the priorities are assigned to the maintenance steps according to the maintenance task, the following property should be preserved:

- (44) If maintenance steps M_q and M_p are intended to implement parts of the same maintenance task, then steps M_q and M_p are assigned the same priority value.

4. Or other forum that includes the user (or his representatives) and the maintenance team management.

In the case of assigning priorities to composite or inducing maintenance steps, the following defines the priorities of the spawned and the induced steps:

- (45) If composed/inducing step M_q is assigned a priority value N , then all maintenance steps M_p that are spawned/induced by the step M_q are assigned the same priority value N .

It should be noted that the priority mechanism should not be misused, e.g., all maintenance steps should not be assigned the same priority. Also, it is advisable to keep the range of the priority values as small as possible without altering their meaning.

In addition to the partial ordering that arises from assigning the priority values to maintenance steps, there may exist additional constraints that impose execution ordering between two or more steps. Such constraints may represent specifics of a maintenance task or inter-step dependencies that cannot be expressed by the input and output relations of the maintenance steps as defined in (13) and (4). The intent of such execution ordering is to impose a sequential rather than a concurrent execution of the maintenance steps due to the above mentioned constraints.

In order to account for the execution ordering ranking, we will introduce the "*precedes*" relation which is defined between the pending maintenance steps as follows:

- (46) If atomic step M_q precedes atomic step M_p , then the step M_q must be implemented before the step M_p .

Because of the semi-atomic nature of induced maintenance steps (see (37) - (43)), the following holds true:

- (47) If inducing step M_q precedes (or is preceded by) step M_p , then all induced steps $M_{q(n)}$ precede (or are preceded by) the step M_p .
- (48) If induced step $M_{q(n)}$ precedes (or is preceded by) step M_p , then its inducing step M_q precedes (or is preceded by) the step M_p .
- (49) An inducing step M_q cannot precede its induced steps $M_{q(n)}$, and vice versa.

Naturally, similar constraints apply to the composite steps and steps that are spawned by them, i.e.,:

- (50) If atomic step M_q precedes composite step M_p , then the step M_q precedes all the maintenance steps spawned (directly or recursively) by the step M_p .
- (51) If composite step M_q precedes step M_p , then all step spawned (directly or recursively) by the step M_q precede the step M_p .

The "precedes" relation is transitive, asymmetric and irreflexive, i.e., the following holds true:

- (52) If step M_q precedes M_r and step M_r precedes M_p , then step M_q precedes step M_p .
- (53) If step M_q precedes M_p , then step M_p cannot precede M_q .
- (54) \forall steps $M_p \in M$, M_p cannot precede itself.

Both the transitive and the asymmetric properties of the "precedes" relation imply that the graph of the "precedes" is acyclic, i.e., the situation in which step M_q precedes M_r , step M_r precedes M_p , and step M_p precedes step M_q is impossible. Also, the situation in which a spawned maintenance step precedes its spawning step is illegal.

Unlike the priority values that suggest the implementation ordering, the "precedes" relation imposes strict ordering between two or more maintenance steps, e.g., in the rule (46) the step M_q will be implemented before the step M_p even if the priority of the step M_p is higher than that of the step M_q . Therefore the ordering imposed by the precedes relation is "stronger" than the one we would obtain using only the priority values of the maintenance steps.

The "precedes" relation must be consistent with the dependency relation between maintenance steps (as it is defined in (13)), since the dependencies that propagate through the primary inputs impose the "precedes" relation between the steps:

- (55) If $\exists C_i \in C$ such that $C_i \in O_{M_q}$ and C_i is the primary input of step M_p , then the step M_q precedes step M_p .

Also, there exist implicit precedence relations that concern inducing maintenance steps (see (37)) which can be stated explicitly as follows:

(56) If \exists inducing step M_q with primary input C_j , then \forall steps $M_p \in M_{C_j}$ M_p precedes M_q .

Recall that the "precedes" relation is not concerned solely with step dependencies that propagate through inputs of a maintenance steps. It deals also with the constraints that are external to the system configuration.

The "precedes" relations between maintenance steps should be defined by the maintenance management, and it is their responsibility not to misuse this mechanism. An incorrect use of the "precedes" relation will lead to introduction of many unnecessary constraints in the scheduling of the maintenances steps. On the other hand, the correct use of the "precedes" relation may improve the effectiveness of the maintenance process. For example, a policy that defines precedence of induced steps over the non-induced steps with the same primary inputs and equal or lower priority, will cause the execution of induced maintenance steps first. This can reduce the effort that would be required later to implement the pending maintenance steps.

5. Future Work and Acknowledgments

The initial goal of this work was to develop a software maintenance model for a Computer Aided Prototyping System (CAPS) [Ref: 5]. We further discovered that no general software maintenance model has been mathematically formalized. This motivated us to work on a more general model for a family of applications since such model will apply to many problems in military software maintenance [Ref: 6]. We are currently working on deriving algorithms for scheduling maintenance tasks using the model described in this paper as well as specific applications to the CAPS system.

We would like to thank Prof. Valdis Berzins, Prof. N. F. Schneidwind, Dr. Bernd Kraemer, Prof. Tarek Abdel-Hamid, and Lt. Laura White (USN) for their time and effort which has substantially improved this paper.

This research was supported in part by the National Science Foundation under grant number CCR-8710737.

List of References

1. Borison E., A Model of Software Manufacture. Advanced Programming Environments, Proceedings of an International Workshop, Trondheim, Norway, June 1986. SPRINGER-VERLAG pp. 197-220.
2. Heimbigner D. and Krane.S, A Graph Transform Model for Configuration Management Environments, Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. 1988. pp. 216-225.
3. IEEE Guide to Software Configuration Management, ANSI/IEEE Std 1042-1987, Technical Committee on Software Engineering of the Computer Society of IEEE, 1988.
4. Martin R.J., Osborne W.M. Guidance on Software Maintenance. National Bureau of Standards, U.S. Departments of Commerce, December 1983. pp. 10-11.
5. Luqi, "Software Evolution through Rapid Prototyping", IEEE Computer, May 1989, pp. 13-25.
6. Mostov I. and Luqi, "Maintenance Problems in Military Software Systems", Technical Report, Computer Science Department, Naval Postgraduate School, NPS52-90-004.

DISTRIBUTION LIST

- | | | |
|------|--|----|
| (1) | Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22304-6145 | 2 |
| (2) | Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, CA 93943 | 2 |
| (3) | Center for Naval Analysis
4401 Ford Avenue
Alexandria, VA 22302-0268 | 1 |
| (4) | Director of Research Administration
Attn: Prof. Howard
Code 012
Naval Postgraduate School
Monterey, CA 93943 | 1 |
| (5) | Chairman, Code 52
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100 | 1 |
| (6) | Chairman, Code MA
Mathematics Department
Naval Postgraduate School
Monterey, CA 93943-5100 | 1 |
| (7) | Naval Postgraduate School
Prof. Luqi, Code CSLq
Computer Science Department
Monterey, CA 93943 | 29 |
| (8) | Naval Postgraduate School
Prof. K. Hefner, Code MAHk
Mathematics Department
Monterey, CA 93943 | 20 |
| (9) | Chief of Naval Research
800 N. Quincy Street
Arlington, Virginia 22217 | 1 |
| (10) | Research Administration
Code 012
Naval Postgraduate School
Monterey, CA 93940 | 1 |
| (11) | Carnegie Mellon University
Software Engineering Institute
Department of Computer Science
Attn. Dr. E. Borison | 1 |

- Pittsburgh, Pennsylvania 15260
- (12) Commanding Officer 1
 Naval Research Laboratory
 Code 5150
 Attn. Dr. Elizabeth Wald
 Washington, D.C. 20375-5000
- (13) Defense Advanced Research Projects Agency (DARPA) 1
 Integrated Strategic Technology Office (ISTO)
 Attn. Dr. B. Boehm
 1400 Wilson Boulevard
 Arlington, Virginia 22209-2308
- (14) Attn: Mr. William McCoy 1
 Code K54, NSWC
 Dahlgren, VA 22448
- (15) Defense Advanced Research Projects Agency (DARPA) 1
 Director, Naval Technology Office
 1400 Wilson Boulevard
 Arlington, Virginia 2209-2308
- (16) Defense Advanced Research Projects Agency (DARPA) 1
 Director, Prototype Projects Office
 1400 Wilson Boulevard
 Arlington, Virginia 2209-2308
- (17) Defense Advanced Research Projects Agency (DARPA) 1
 Director, Tactical Technology Office
 1400 Wilson Boulevard
 Arlington, Virginia 2209-2308
- (18) Chief of Naval Operations 1
 Attn: Dr. R. M. Carroll (OP-01B2)
 Washington, DC 20350
- (19) Dr. Aimram Yehudai 1
 Tel Aviv University
 School of Mathematical Sciences
 Department of Computer Science
 Tel Aviv, Israel 69978
- (20) Dr. Robert M. Balzer 1
 USC-Information Sciences Institute
 4676 Admiralty Way
 Suite 1001
 Marina del Ray, California 90292-6695
- (21) Editor-in-Chief, IEEE Software 1
 Attn. Dr. Ted Lewis
 Oregon State University
 Computer Science Department
 Corvallis, Oregon 97331

- (22) IBM T. J. Watson Research Center 1
 Attn. Dr. A. Stoyenko
 P.O. Box 704
 Yorktown Heights, New York 10598
- (23) International Software Systems Inc. 1
 12710 Research Boulevard, Suite 301
 Attn. Dr. R. T. Yeh
 Austin, Texas 78759
- (24) Kestrel Institute 1
 Attn. Dr. C. Green
 1801 Page Mill Road
 Palo Alto, California 94304
- (25) Prof. D. Berry 1
 Department of Computer Science
 University of California
 Los Angeles, CA 90024
- (26) MCC AI Laboratory 1
 Attn. Dr. Michael Gray
 3500 West Balcones Center Drive
 Austin, Texas 78759
- (27) National Science Foundation 1
 Division of Computer and Computation Research
 Attn. Tom Keenan
 Washington, D.C. 20550
- (28) Naval Ocean Systems Center 1
 Attn: Linwood Sutton, Code 423
 San Diego, California 92152-5000
- (29) Naval Ocean Systems Center 1
 Attn. Les Anderson, Code 413
 San Diego, California 92152-5000
- (30) Naval Sea Systems Command 1
 Attn: CAPT A. Thompson
 National Center #2, Suite 7N06
 Washington, D.C. 22202
- (31) NAVSEA, PMS-4123H 1
 Attn. William Wilder
 Arlington, VA 22202-5101
- (32) New Jersey Institute of Technology 1
 Computer Science Department
 Attn. Dr. Peter Ng
 Newark, New Jersey 07102
- (33) Office of Naval Research 1
 Computer Science Division, Code 1133
 Attn. Dr. Van Tilborg

- 800 N. Quincy Street
Arlington, Virginia 22217-5000
- (34) Office of Naval Research 1
Computer Science Division, Code 1133
Attn. Dr. R. Wachter
800 N. Quincy Street
Arlington, Virginia 22217-5000
- (35) Office of Naval Research 1
Applied Mathematics and Computer Science
Attn. J. Smith, Code 1211
800 N. Quincy Street
Arlington, Virginia 22217-5000
- (36) Software Group, MCC 1
9430 Research Boulevard
Attn. Dr. L. Belady
Austin, Texas 78759
- (37) University of California at Berkeley 1
Department of Electrical Engineering and Compute Science
Computer Science Division
Attn. Dr. C.V. Ramamoorthy
Berkeley, California 90024
- (38) Attn: Dr. Mike Reiley 1
Fleet Combat Directional Systems Support Activity
San Diego, CA 92147-5081
- (39) Chief of Naval Operations 1
Attn. Dr. Earl Chavis (OP-162)
Washington, DC 20350
- (40) Steve Huseth 1
Honeywell Systems & Research Center
Mpls, MN 55418
- (41) Attn: George Sumiall 1
US Army Headquarters
CECOM
AMSEL-RD-SE-AST-SE
Fort Monmouth, NJ 07703-5000
- (42) Attn: Joel Trimble 1
1211 South Fern Street, C107
Arlington, VA 22202
- (43) Attn: Dr. David Hislop 1
United States Laboratory Command
Army Research Office
P. O. Box 12211
Research Triangle Park, NC 27709-2211

- (44) Attn: Dr. Phil Hwang 1
NSWC, U-33
Silver Spring, MD 20903-5000
- (45) Attn: Dr. Abraham Waksman 1
Computer Science and Artificial Intelligence
Department of the Air Force
Bolling Air Force Base, DC 20332-6448
- (46) Israeli Air Attache 2
Israeli Embassy
3514 International Drive
Washington, DC 20008