The Abstraction Refinement Model and The Modification-Cost Problem*

Benjamin J. Keller and Richard E. Nance

TR 92-29

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

May 22, 1992

**Also cross-listed as Systems Research Center report SRC-92-002. This work was partially supported by the Naval Surface Warfare Center, White Oak Detachment, through contract number N60921-89-D-A239-0015-01.*

Abstract

A problem common to systems and software engineering is that of estimating the cost of making changes to a system. For system modifications that include changes to the design history of the system this is the "modification-cost" problem. A solution to this problem is important to planning changes in large systems engineering projects.

A cost model based on the Abstraction Refinement Model (ARM) is proposed as a framework for deriving solutions to the modification-cost problem. The ARM is a characterization of software evolution that is also appplicable to general systems. Modifications to systems and their design histories are described using the components of the ARM. The cost model is defined by functions on the ARM components. The derived solution is given by an abstract expression of the cost functions.

CR Categories & Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance – Corrections, Documentation, Enhancement; D.2.9 [**Software Engineering**]: Management – Cost estimation, Life cycle

Additional Keywords & Phrases: Systems Engineering, Design, Modification, Software Evolution, Model.

CONTENTS

1. Introduction

A modification to a complex system consisting of software, hardware, and humanware can affect all three types of components and shift responsibilities among them. Clearly, the cost of making modifications to existing complex systems is often very high; therefore, prediction of the cost of a modification before its application is essential. This problem of prediction, or estimation of the cost of a modification, is called the "modification-cost problem."

The modification-cost problem for systems engineering is the following:

- Given a system with its complete set of documentation, and an intended modification of the system or its documentation,
- Estimate the cost of making the change induced by the modification such that a new system implementation is found and a new documentation set is formed.

In this problem, the complete set of documentation for a system is all documentation from requirements to implementation that describes the characteristics of the system (i.e. a 'design history').

This report proposes a solution through the development of a model for estimating the cost of the intended change. This cost model is based on the Abstraction Refinement Model (ARM), a model of software evolution [Keller and Nance, 1992]. The ARM, used here for description of general system evolution, is extended to reflect a different perspective on the activities of system modification.

This solution assumes a similarity between software and systems engineering that legitimizes the use of the ARM. For complex systems, the software engineering domain is embedded in that of systems engineering. That fact does not imply that problems specific to software can be generalized to systems engineering. However, as illustrated by this report, the modification-cost problem is one that can be stated analogously for both software and general systems. Both the systems and software engineering fields undoubtedly benefit from the identification of such common problems and their solutions. This work is encouraged by related efforts noting the applicability of software engineering models and frameworks to systems engineering (see [Krieder and Nance, 1991]).

Organizationally, the following section (Section 2) defines the ARM and describes its use as a system evolution process model. Section 3 defines the cost model built upon the ARM, and Section 4 provides a discussion of the possible extensions of the cost model, and further directions.

2. Abstraction Refinement Model

The Abstraction Refinement Model (ARM) is a model of the software evolution process. The

model is first described as a means for characterizing the role of reverse engineering in software maintenance activities [Nance et al., 1989]. The formal development of the ARM as a model of software evolution is given in [Keller, 1990], and a less formal presentation appears in [Keller and Nance, 1992]. The ARM is interpreted in the systems engineering domain below.

The cost model developed in this report is based on extensions to the ARM. Intuitively, the ARM consists of three types of structures: system descriptions, system description transformations, and an abstraction relation on the system descriptions. The system descriptions are products such as requirements, designs or programs. The transformations when applied to these products transform them into a different product which is possibly less abstract (or more refined). The abstraction relation indicates whether a pair of system descriptions are related in the sense that one is a realization of the other.

Discussions of the ARM typically use the term 'specification' to mean a system description at any level of abstraction. However, the term is more likely to be relative: for a pair of system descriptions related by the abstraction relation, the more abstract is a *specification* of the other. Also, if a is a specification of b then b is a *realization* of a.

The components of the ARM form the primary interest of the model, and are given more detailed definitions below. A description of how these components can be used to represent the process of software evolution establishes the foundation for the extension to system evolution.

2.1. Definitions.

The three basic components of the ARM have more formal representation in the model. Formally:

(1) system descriptions are elements of languages,
(2) transformations are mappings on these languages, and
(3) the abstraction relation is a preorder on the elements of these languages (a preorder is a reflexive and transitive relation).

The ARM is a collection of languages, called a language system, and a collection of transformations. (The definition of the ARM here, is more general than that given in [Keller, 1990].)

A *language system* is a family of languages and a collection of preorders on the languages; so that elements of one language can be related to elements of the others.[1] (Language systems are discussed briefly in [Keller, 1990], but the details of their definition are unimportant for the purposes of this

[1]For complex systems, a 'language' would consist of sublanguages for software, hardware and humanware.

report.) The interpretation of the preorders is that if $a \sqsubseteq b$, then a is a correct realization of b. The act of verifying that a realizes b, is a "proof" that $a \sqsubseteq b$.

The preorders give the language system a structure based on abstraction, and for the purposes of this study the distinctions among languages are non-essential. Therefore, the language system is considered to be a set L with lattice-like structure. This defines the "space" on which systems can be defined, developed, and maintained. Note that these three terms are typically associated with successive periods in the life of a software system: definition (requirements), development (design, implementation, test) and maintenance. Coupling the three leads to the more descriptive term "software evolution."

Symbolically, the language system is the pair $\langle L, \sqsubseteq \rangle$ where L is a "language," and $\sqsubseteq$ is a preorder on L, meaning $\sqsubseteq$ is a relation on L such that

(1) for all $l \in L, l \sqsubseteq l$ (reflexive)
(2) if $a \sqsubseteq b$ and $b \sqsubseteq c$ then $a \sqsubseteq c$, for $a, b, c \in L$ (transitive).

The component L could be described as a wide-spectrum language [Neighbors, 1984; Keller, 1990], i.e. a language in which products from requirements through implementation can be expressed.

Transformations are mappings on this structure. A *transformation* from a to b, denoted $t : a \mapsto b$, is a mapping $t : L \rightarrow L$ such that $t(a) = b$. This definition means that any mapping that takes a to b is suitable to serve as the transformation from a to b. The collection of transformations is denoted T, and defined as $T = \{t : a \mapsto b | a, b \in L, t : L \rightarrow L\}$.

The ARM is the pair $\langle \langle L, \sqsubseteq \rangle, T \rangle$, where $\langle L, \sqsubseteq \rangle$ is a language system (with one language) and T is the set of transformations on L. A language system and set of transformations is sufficient to describe software evolution.

As indicated above, the abstraction relation, the preorder $\sqsubseteq$, models the correctness relationship between pairs of system descriptions. Using this order we can define the sets of realizations for a specification. For an arbitrary set of system descriptions $S \subseteq L$ and a system description $s \in L$, the collection of realizations of s in S is $S(s] = \{p \in S | p \sqsubseteq s\}$. Similarly, the set of specifications of s in S is $S[s) = \{p \in S | p \sqsupseteq s\}$. So for any system description $s \in L$, its specifications are $L[s)$ and its realizations are the members of $L(s]$.

The collection of realizations of a system description s could be depicted as in Figure 1. Each node in the diagram represents a system description, and the node at the top represents s. The lines show the abstraction relation: if a node a is above and connected to a node b by a line then

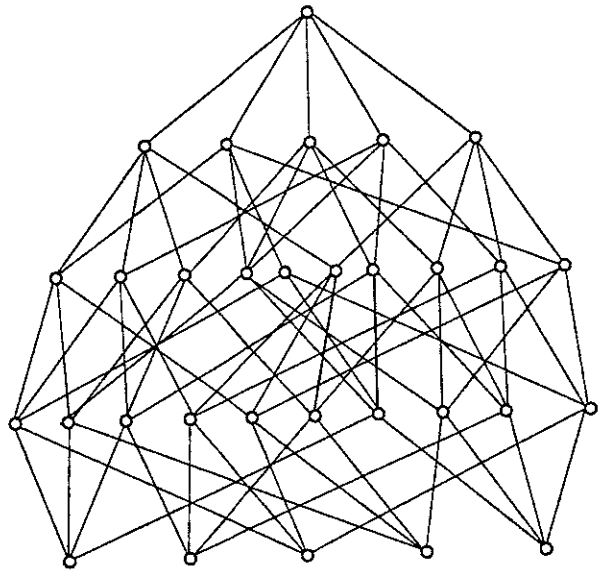


Figure 1. Abstraction Structure of the ARM

the system description a is a specification of the system description b. These diagrams are intuitive representations of the model, and should not be confused with the model itself. The structure of the diagram is based on the assumptions about $\langle L, \sqsubseteq \rangle$ in [Keller, 1990, p.46]. These assumptions have not been shown to be applicable to arbitrary language systems and are not used here other than for the diagrams.

Thus far, the relationship between the abstraction relation and the transformations has been ignored. Generally, transformations model an action on a system description either in a forward or reverse sense. A transformation t that maps a to b is called *forward* if $a \sqsupseteq b$, and *reverse* if $a \sqsubseteq b$. Note that a transformation need be neither correctness preserving ($a \sqsupseteq b$) nor directed (forward or reverse).

In the discussion of software evolution, and software development in particular, the notion of a *system context* is important. Informally, this is the set of documentation produced during the development of a system (with a few provisos). Formally, the *context* from a system description s is a pair $\langle SD_s, T_s \rangle$, where

(1) $SD_s = \{s_i \in L | s = s_0, 0 \leq i \leq n\}$ is a set of system descriptions, and
(2) $T_s = \{t : s_i \mapsto s_{i+1} | 0 \leq i < n\}$ is a set of transformations on SD_s.

Ideally, a context is a "chain" of system descriptions and forward transformations such that $s \mapsto s_1 \mapsto s_2 \ldots s_{n-1} \mapsto s_n$ and $s \sqsupseteq s_1, s_1 \sqsupseteq s_2, \ldots, s_2 \sqsupseteq s_{n-1}$, and $s_{n-1} \sqsupseteq s_n$.

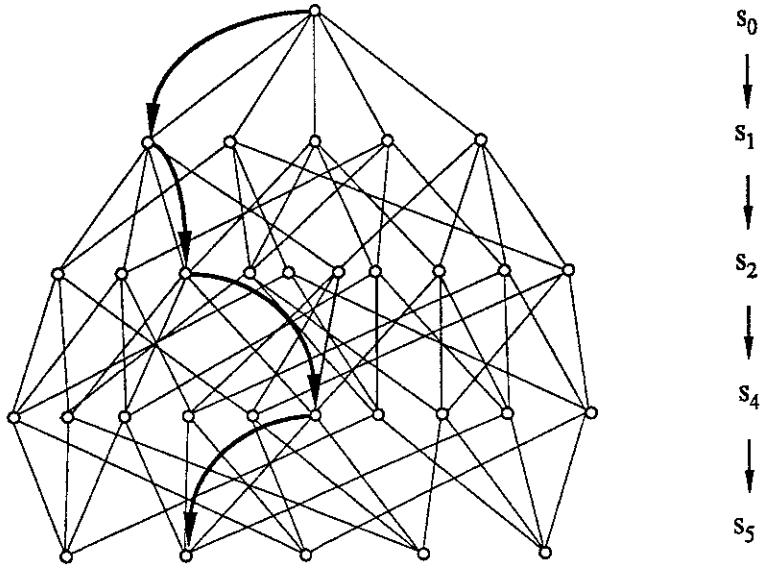


Figure 2. Diagrams of a Systems Context in the ARM: (a) in Reference to the Abstraction Structure, and (b) Alone

The context of a system s can be diagrammed in two ways. Figure 2(a) shows the context with respect to the rest of the model, and Figure 2(b), with just the system descriptions and transformations.

Recall the notation for realizations. For contexts, $SD_s(p]$ is the set of realizations of p in SD_s. Sometimes, it will be necessary to denote the system descriptions "below" some element $p \in SD_s$ in a context $C = \langle SD_s, T_s \rangle$. This is denoted $C{\downarrow_p}$ and is $\langle SD_s{\downarrow_p}, T_s{\downarrow_p} \rangle$, where

(1) $SD_s{\downarrow_p} = \{ s_i \in SD_s | t : p \mapsto s_i, t \in cl(T_s) \}$, where $cl(T_s)$ is the collection of functions formed by the composition of transformations in T_s, and
(2) $T_s{\downarrow_p}$ is the set $\{ t \in T_s | \exists s_0, s_1 \in SD_s{\downarrow_p}, t : s_0 \mapsto s_1 \}$.

The distinction between $SD_s(p]$ and $SD_s{\downarrow_p}$ is that the first is the collection of realizations of p in the context and the second is the collection of system descriptions in the context found by transformations on p. Clearly, we want these sets to be identical, and this is the ideal context described above. In this situation, the condition $SD_s{\downarrow_p} = SD_s(p]$ holds for all $p \in SD_s$. Alternatively, this is the requirement that the transformations in the context yield correct results.

2.2. Process Model.

The role of the ARM as a model of the software evolution process is detailed elsewhere [Keller

and Nance, 1992] and briefly described above. The elements of L are the software objects on which the process acts. The preorder $\sqsubseteq$ on L models correctness between system descriptions, and the elements of T represent the process actions. These combine freely to describe many different approaches to software evolution in terms of the underlying process.

Diagrams play an important role in the visualization and understanding of the ARM. The two diagrammatic forms used are shown in Figure 2. We caution that these diagrams are not the model itself but a graphical representation of the model. (In addition to these diagrammatic forms, we also use *commutative diagrams*. These diagrams are directed graphs with nodes representing objects of some kind and edges representing "mappings" on the objects. A diagram commutes when the "mappings" represented by two paths from the same source node to the same target node are equal. Essentially, a commutative diagram is a graphical representation of an equation. For more detail see [Barr and Wells, 1990, pp. 76–82].)

2.2.1. Product Construction (Development).

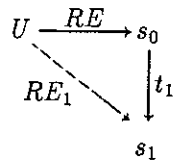
The "evolution" of a software product begins with definition. System definition as the precursor to development is realized through activities that could be described as requirements engineering, which takes the "customer's" view of the product and transforms it into the "developer's" view as shown in the following diagram.

$$U \xrightarrow{RE} s_0$$

The developer's view is essentially the requirements specification for the product. Alternatively, the result of the requirements engineering activities is the context $\langle \{s_0\}, \emptyset \rangle$.

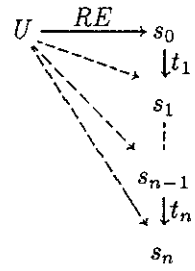
Note that the ARM provides a partial representation of the requirements engineering activities. The model only represents the space of developer's views. Therefore, only the effect of requirements engineering can be represented, i.e. the selection of the system description s_0 from L based on the external U.

Once the requirements are stated in the developer's terms (i.e. as an expression in L), refinement can begin. Refinement is represented as transforming s_0 to some s_1 by a transformation t_1, as shown by



This transformation changes the context to $\langle\{s_0, s_1\}, \{t_1\}\rangle$ (in the terminology of [Keller and Nance, 1992] t_1 *narrows* the context). The dashed arrow is intended to show an implicit requirements engineering activity produced by first doing RE and then t_1.

Following the completion of development, a context $\langle\{s_0, \ldots, s_n\}, \{t_1, \ldots, t_n\}\rangle$ has been constructed with the diagram



(1)

where s_n is the implementation-level system. One possible realization of diagram (1) in terms of the model is shown in Figure 2(a).

Note the apparent (or implicit) assumption in this representation that a purely top-down approach is being taken. This is not necessarily the case. The ARM is capable of representing rapid prototyping [Keller, 1990] and in fact diagram (1) is very similar to the description of development through rapid prototyping by Rattray and Price [1990, p. 96]. The intended structure of a context, however, does correspond to that produced by a top-down approach (the system descriptions form a chain with respect to the abstraction order). Nevertheless, an iterative process could result in a context which gives the appearance of a top-down approach (consider how proofs in mathematics are first developed and then later refined). (See [Parnas and Clements, 1985; Lano and Haughton, 1991].) Therefore, the assumption that a context is created by the development activity places no constraints on how this developmental activity is performed.

2.2.2. Product Change (Maintenance).

The activities that effect changes in a software system are generally termed "maintenance." These activities contribute to the evolution of the system. The description of maintenance given here emphasizes how a modification might be approached, differing from [Keller and Nance, 1992] which emphasizes description of maintenance forms.

A change initiates with a modification request from the 'customer' to be applied to a software system with an existing context. The modification request is effectively a transformation MR on the customer's view of the system, that is to be paralleled by a change in the developer's view of

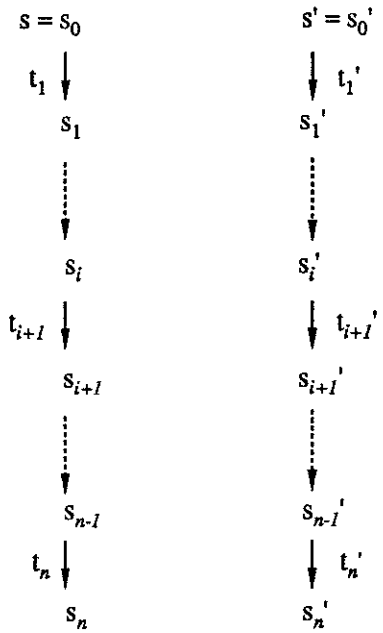


Figure 3. Current and Objective Contexts

the system. This perception leads to the diagram

$$\begin{array}{ccc} U & \xrightarrow{MR} & U' \\ {\scriptstyle RE}\downarrow & & \downarrow{\scriptstyle RE'} \\ C & \xrightarrow[M]{} & C' \end{array}$$

which indicates that the effect of making the modification M should be *the same* as beginning over with the customer's revised view and redeveloping [Yau, 1984, p. 14].

The pervading question is: How is M to be produced from transformations on the elements of the context? The situation is described by the diagram in Figure 3, where the left side is the context C and the right side is the context C'.[2] How a change is represented depends on the objective of the change. If the objective is just to produce a new implementation s'_n then the context need not be preserved, but if the objective is to produce a new implementation *in its context*, then the context must be preserved.

If context preservation is not an objective then the general form of a change is given by Figure 4. This diagram shows (1) a horizontal transformation making the change, and (2) forward transformations producing the new implementation. If the initial horizontal transformation is made at the implementation level, the need for the refinements is eliminated.

[2]It is not necessary that C and C' contain the same number of system descriptions despite the obvious interpretation of Figure 3. C' can be made smaller than C by letting $s'_i = s'_{i+1}$ for some $0 \leq i < n$, and larger by introducing "invisible" system descriptions between s'_i and s'_{i+1}.

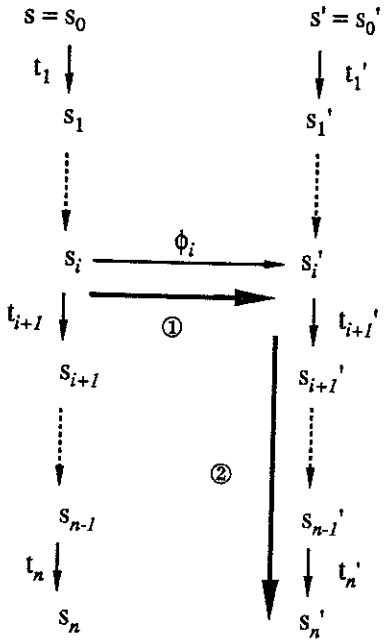


Figure 4. Non-Context Preserving Change

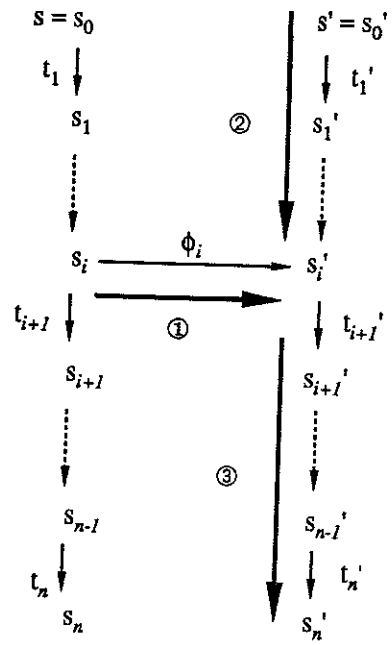


Figure 5. Context Preserving Change

Redevelopment also gives an alternative that is not truly context preserving. The approach is to scrap the existing system and its context and to begin over. Such an approach produces an entirely new context and system which is unrelated to the existing context (in practice, this is probably never accomplished so cleanly).

For context preserving approaches, the challenge is to make a change at some appropriate level and then propagate the change throughout the context, shown in Figure 5. As before, (1) a horizontal change is made, and (2) the result is refined to a new implementation. However the context above the change must be made consistent with the rest (3).

This form of change corresponds in some ways to the approach described for maintenance using the ARM [Keller, 1990; Keller and Nance, 1992]. That approach casts the goal as finding the least common abstraction of s_n and s'_n (this is a system description α such that α is the least element of L that satisfies $s_n \sqsubseteq \alpha$ and $s'_n \sqsubseteq \alpha$) denoted $s_n \sqcup s'_n$, and the least common context element.[3] The task is to complete the context from the least common context element to $s_n \sqcup s'_n$, and refine $s_n \sqcup s'_n$ to s'_n (see [Keller and Nance, 1992] for more details).

[3]In the generalization of the ARM a least common abstraction of arbitrary system descriptions is not guaranteed to exist.

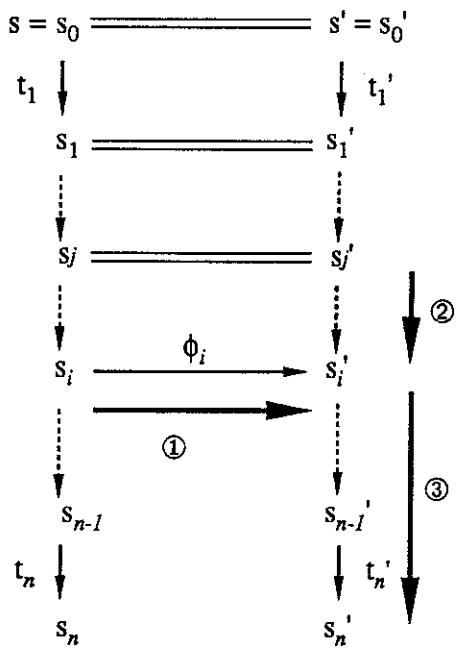


Figure 6. Change with Coinciding Contexts

The approach here is slightly more general (and possibly more feasible). The change is made at some level of abstraction on the system description s_i (see Figure 6). The change is made (1) using a transformation ϕ_0 that takes s_i to the new system s_i'. Then (2) an ancestor of $s_n \sqcup s_n'$ is found in the original context, $s_j = s_j'$ in Figure 6, above which the contexts coincide. The context must then be completed from s_j to s_i'. Finally, (3) the change must be propagated to a new implementation s_n'.

A possible complication in the second step is that $s_n \sqcup s_n'$ has no ancestor in the context. In this case, s_i is the wrong choice for the change and in fact the change should be made to s_0 since the contexts do not coincide, shown in Figure 7. The change represented here is too major (or too "abstract") to be made at the level of s_i. In general, finding the level "closest" to the least common context element reduces the effort involved in the context completion step (2). The case where contexts do not coincide is best approached by changing the highest level as shown in Figure 8.

Note that propagating changes (the third activity in all diagrams) does not necessarily mean discarding the context below the change. A change, a horizontal transformation, may add, delete or modify the components of the system description to which it is applied. Provided sufficient traceability is included in the context, the changes can be propagated through the realizations of the system.

The idea of change propagation may be thought of in a different way. A part of the original

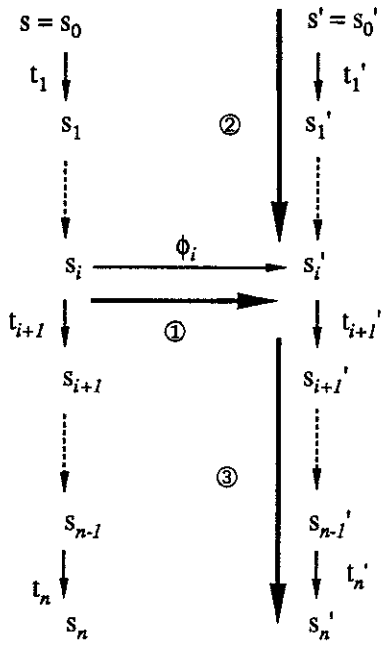


Figure 7. Improper Change with Non-coinciding Contexts

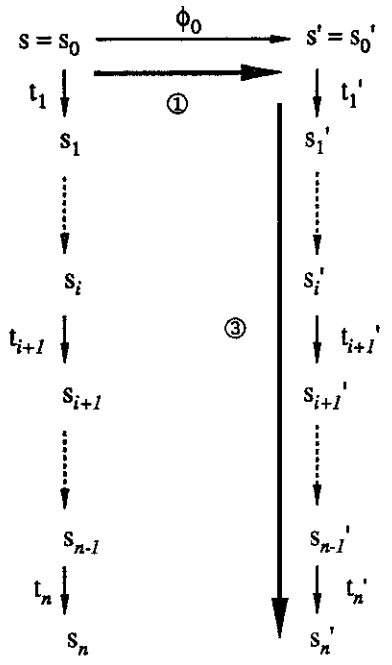


Figure 8. Proper Change with Non-coinciding Contexts

context with the change ϕ_0 applied to s_i is represented by the diagram

$$\begin{array}{ccc} s_i & \xrightarrow{\phi_0} & s_i' \\ {\scriptstyle t_{i+1}}\downarrow & & \\ s_{i+1} & & \end{array} \tag{2}$$

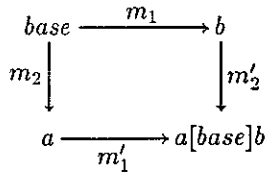
In order to find the completion of the new context for diagram (2), we need to find a s'_{i+1} and transformation t'_{i+1} that takes s'_i to s'_{i+1}. This is shown by the diagram

$$\begin{array}{ccc} s_i & \xrightarrow{\phi_0} & s_i' \\ {\scriptstyle t_{i+1}}\downarrow & & \downarrow{\scriptstyle t'_{i+1}} \\ s_{i+1} & \xrightarrow[\phi_1]{} & s'_{i+1} \end{array} \tag{3}$$

In mathematics, diagram (3) is called a *pushout* (from category theory, see Barr and Wells [1990] for an introduction).

Diagram (3) is also very similar to one for the integration problem addressed by the HPR-algorithm [Reps, 1991]. The integration problem is: given an original program *base* and two programs

a and b that are modifications of *base*, find a program $a[base]b$ that integrates a and b so that the changes are consistent. This is similarly diagrammed as



where m_1, m_1', m_2 and m_2' are modifications on programs. Taking

(1) s_i to be the base,
(2) ϕ_0 to be m_1,
(3) t_{i+1} to be m_2,
(4) s_{i+1} to be a and
(5) s_i' to be b,

the task of making the change is essentially an integration problem. This suggests that the HPR-algorithm may give a suitable tool for propagating changes in a context. Currently, however, the algorithm is only usable for programs in a restricted language.

Despite the limitation of the algorithm, the analogy between integration and change propagation suggests another approach to making changes that is shown in Figure 9. In this approach, (1) the change is introduced through a horizontal transformation, and then propagated in both directions, (2) and (3), through the context. (Note that the upward propagation does not have the same structure as the downward and may not be performed in the same way.)

Context changes in which the context is preserved have been described in several ways. However each consists of three subordinate activities: (1) a horizontal transformation representing the change, (2) propagation of the change upward in the context, and (3) propagation of the change downward in the context. When the context is not preserved, upward propagation is ignored.

The description of maintenance given above focuses on individual maintenance tasks. In the characterization, each modification takes a context C to a context C'. C' becomes the current context, but it is not clear what happens to C. The two possibilities are that

(1) C is forgotten, or
(2) C is retained through some mechanism not represented by the model.

The choice between the two depends on what information is necessary for continued maintenance[4] and is not relevant to this report.

[4]The set of documentation needed for maintenance is discussed by Nance et al. [1989, p. 24].

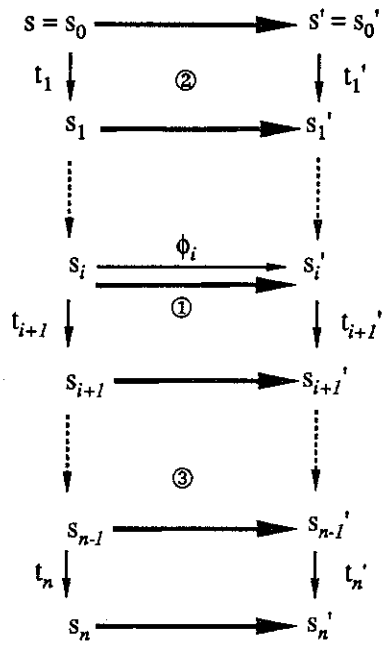


Figure 9. Complete Context Preserving Change

2.3. Abstraction Refinement and Systems Engineering.

Although the ARM is a model of software evolution, the general form presented here can be extended naturally to the systems engineering domain by interpreting system descriptions as combinations of specifications of the software, hardware and humanware components of the system. The abstraction order and transformations can be defined in terms of abstraction orders and transformations on the software, hardware and humanware domains. Using this interpretation, the ARM is a model of the *system evolution* process similar to that described for software.

3. Cost Model

The modification-cost model is developed from the Abstraction Refinement Model, described in the prior section. The intuition behind the cost model is given first in order to motivate how the ARM is used as a basis. Then the cost functions are defined on the ARM components, and, finally, the functions are used to consider the cost of changes.

3.1. Intuition.

To develop the cost model from the ARM, an understanding of how activities in the system engineering domain are represented in the model is necessary. The ARM components are the system descriptions (L), the abstraction order ($\sqsubseteq$), and the transformations (T). These components depict the system evolution process by treating the transformations as actions on the system descriptions

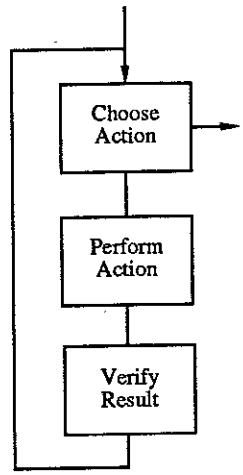


Figure 10. Design Step in the ARM

and the order as the means for verifying correctness of the result.

A design step in the system engineering domain can be rather complex, involving determination and evaluation of alternatives before choosing and executing an alternative. In the ARM this step is much simpler. Starting with a system description, a transformation is selected, the transformation is applied, and the result verified. These three steps generate a system description, to which the steps are iteratively applied until some stopping criterion is met. Figure 10 shows this iterative process.[5] This is essentially the "basic design step" described by Lehman et al. [1984, pp. 42–44], and the "evolutionary process" of design of Dasgupta [1991, p. 77].

The three steps: (1) select action, (2) perform action, and (3) verify result, engender three costs:

(1) select action: the cost of determining which transformation to apply (i.e. making a design decision),
(2) perform action: the cost of applying the transformation, and
(3) verify result: the cost of assuring that the action taken produces the desired result.

At this point, these three components are believed to characterize sufficiently the cost of a software development or change process.

3.2. Formal Definition.

The basis for each of the cost functions is described in the following paragraphs. The discussion

[5]This diagram is a representation of the three steps in the ARM, and is by nature "information-deficient." A methodology should provide information on how these steps are to be achieved (see [Krieder and Nance, 1991]).

of costs is limited to the implications for the representation with respect to the ARM. The basis for cost value assignment is treated in Section 4.

3.2.1. Cost of Selection.

The cost of making a design decision is the cost of selecting a transformation. This cost function is defined as a mapping $\chi : L \rightarrow \mathbf{R}$ that assigns real-valued costs to the system descriptions. So $\chi(a)$ for $a \in L$, is the cost of deciding how to proceed from the system description a. This cost is assumed to be independent of the last transformation applied; it is dependent only on the current system description.

Intuitively, this cost is somehow determined by the factors:

(1) The "location" of a in the language system $\langle L, \sqsubseteq \rangle$ (i.e. how many realizations of a are there in L, or what is $|L(a]|$?).[6]
(2) The extent to which the methodology limits the selection of transformations.
(3) The extent to which the methodology aids the selection of transformations.

The first two factors are related: the number of transformations possible from a is $|L(a]|$. So if the methodology limits the selection in some way, the choice is bounded by the number of realizations. The third factor depends on the methodology providing means to make the selection; i.e. evaluative guidance in comparing alternatives.

3.2.2. Cost of Transformation.

The cost of a transformation is defined by a mapping $\tau : T \rightarrow \mathbf{R}$. Recall that transformations are defined as mappings that take a source to some target system description, so each transformation is associated with a source-target pair. Thus when τ is applied to $t : a \mapsto b$, $\tau(t)$ is the cost of finding b from a with t. Two transformations $t : a \mapsto b$ and $t' : a \mapsto b$ can exist such that $\tau(t) \neq \tau(t')$.

The cost function is assumed to meet the following conditions:

(1) The cost of performing a transformation depends solely on the system description to which it is applied.
(2) The cost of the composition of two transformation is the sum of their costs.

The second assumption, $\tau(t \circ t') = \tau(t) + \tau(t')$, allows the definition of the cost function to be completed by assigning costs to the "basic" transformations (i.e. those that cannot be decomposed).

In justification of (1) above, the transformations $t : a \mapsto b$ and $t' : b \mapsto c$ are dependent in the sense that t' depends on b, the result of t. It might be argued that if t is misapplied it

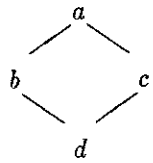
[6] For a set A, $|A|$ is the number of elements in A.

affects the application of t'. However, this is not the way the ARM represents the process. The application of t corresponds to some action in reality. If the action in reality is misapplied, then some other transformation $t_1 : a \mapsto b'$ results (in the model) and t' cannot be applied unless $b = b'$. A transformation $t_1' : b' \mapsto c$, might exist and permit recovery, but the cost of the mishap is accounted for by $\tau(t_1) + \tau(t_1')$ and not $\tau(t) + \tau(t')$. Note that independence among transformations *suggests* the additivity of transformation costs.

3.2.3. Cost of Verification.

The cost of verification is defined by a mapping $\nu : L \times L \rightarrow \mathbf{R}$. The cost of verifying "b is a realization of a," i.e. $a \sqsupseteq b$, is $\nu(a, b)$ which is a constant regardless of the steps taken to find b from a. (Note that in the case of a methodology that enforces the use of correctness preserving transformations consequently having no explicit verification step, this cost is zero).

Consider the diagram



(meaning $a \sqsupseteq b, a \sqsupseteq c, b \sqsupseteq d, c \sqsupseteq d$). In this case, the cost of verifying $a \sqsupseteq d$ is $\nu(a, d) = \nu(a, c) + \nu(c, d) = \nu(a, b) + \nu(b, d)$. A generalization of this rule is the following: for the chain $a = b_0 \sqsupseteq \cdots \sqsupseteq b_n = d$ the cost of verification is $\nu(a, d) = \sum_{i=0}^{n-1} \nu(b_i, b_{i+1})$.

3.2.4. Combining Costs.

The basic process step is made up of the activities *select action*, *perform action*, and *verify result*. The cost of performing such a step from system description a to system description b (i.e. the cost of implementing a by b) is $\chi(a) + \tau(t) + \nu(a, b)$, the sum of the costs of selecting a transformation $t : a \mapsto b$, applying t, and verifying b. The "cost" of a context is given by the cost of its components. The cost of implementing a by d using the chain (context) $a = b_0 \sqsupseteq \cdots \sqsupseteq b_n = d$ and composite transformation $t : a \mapsto d$ is

$$\tau(t) + \nu(a, d) + \sum_{i=0}^{n-1} \chi(b_i).$$

This is probably not the actual cost of the implementation steps taken from a to b since the context may only reflect the "logical structure" of the development and not the more probable iterative path. (This value is probably a lower bound on the actual cost).

3.3. Cost of Change.

The cost model is intended to address the estimation of the cost of a change. Below, this problem is restated and explored using the ARM as a process model.

The ARM characterization of changes, given in Section 2, focuses on the changes with respect to the system context. The general form of such a change, shown in Figure 5, consists of (1) a transformation ϕ_0 which makes the change, (2) propagation of the change up the new context, and (3) propagation of the change down the new context. The modification-cost problem needs to be rephrased in terms of this general form.

PROBLEM 1 (MODIFICATION-COST). *Given a context $C = \langle SD_s, T_s \rangle$ with implementation s_n, and a modification ϕ_0 from $s_i \in SD_s$ to $s_i' \notin SD_s$, determine the cost of making the change induced by ϕ_0 such that:*

(1) *a new implementation s_n', $s_i' \sqsupseteq s_n'$, is found, and*
(2) *a new context $C' = \langle SD_s', T_s' \rangle$ is formed where $s_i', s_n' \in SD_s'$.*

If the context is not of concern, then (2) can be omitted.

One possible solution is given in terms of the characterization. The cost of making the change is the cost of propagating the change up and down the context. Of course, if context preservation is not a goal, the cost of upward propagation is not an issue.

Formally, the cost of the change is

$$\tau(\phi_0) + \mathcal{C}(C'\downarrow_{s_i'}) + \mathcal{C}(C' \setminus C'\downarrow_{s_i'}) \tag{4}$$

where

(1) $\tau(\phi_0)$ is the cost of the initial modification,
(2) $\mathcal{C}(C'\downarrow_{s_i'})$ is the cost of propagating the modification downward, and
(3) $\mathcal{C}(C' \setminus C'\downarrow_{s_i'})$ is the cost of propagating the modification upward.

Each of these costs is considered individually in the following sections.

3.3.1. Cost of Modification.

The initial modification is represented by a transformation ϕ_0 which takes s_i to s_i'. The cost of ϕ_0 is given by $\tau(\phi_0)$. This section presents a classification of such transformations, and then discusses some factors that might influence their performance and cost.

Although not required, we assume that a modification does not have a target at a different level of abstraction. In the ARM this is called a horizontal transformation. Let $\phi_0 : s_i \mapsto s_i'$ be a horizontal transformation and consider the possible actions of ϕ_0.

If $s_i \equiv s_i'$, then s_i and s_i' are equivalent in terms of correctness (i.e. the modification was probably motivated by some other objective). In this case the modification can be made using "algebraic" rules (like the laws of programming [Hoare et al., 1987]) to manipulate the description. The difficulty of making such a modification can vary greatly, but the costs could be determined for each rule.

The remaining forms of transformations ϕ_0 produce an s_i' such that $s_i \not\equiv s_i'$. These are transformations that do one or any of the following to the components of a system description:

(1) delete components,
(2) add components, and
(3) modify components.

In general, a modification causes the deletion, addition, and alteration of several components of s_i. Such a modification should correspond to a transformation that can be decomposed into transformations for each deletion, addition or alteration. Thus, the cost of this type of transformation might be decomposed into the costs of the component deletions, additions, and alterations. (Additional discussion of modifications can be found in [Ramalingam and Reps, 1991].)

Earlier, the existence of an "appropriate" level for a modification is discussed. While this level is not defined formally, it is the level at which the modification is most "natural."[7] Clearly, the ease with which a transformation is made affects its cost; therefore, the level of the transformation is a factor. (Note that the ARM does not represent the activity of finding a modification from a modification request. Finding the "appropriate" level for modification is part of this task, the cost of which is not included in this model.)

3.3.2. Cost of Upward Propagation.

Similar to the case for modifications, the cost of propagating the change upward through the context is affected by the approach taken. Unfortunately, this task is not very well understood, so the approaches available are limited. The approach considered here is that discussed earlier in the presentation of the ARM.

This approach begins with finding an element s of the original context such that the least

[7] A more formal meaning of "appropriate" level in terms of contexts is discussed below

common abstraction of s_i and s_i' is its realization (i.e. $s \sqsupseteq s_i \sqcup s_i'$). Then starting with s, completing the context through $s_i \sqcup s_i'$ to s_i'. This divides the task into the following steps:

ALGORITHM 1.

(1) *compute* $s_i \sqcup s_i'$,
(2) *search for* $s \in C$ *(the original context) such that* $s \sqsupseteq s_i \sqcup s_i'$,
(3) *complete the context from* s *to* $s_i \sqcup s_i'$, *and*
(4) *complete the context from* $s_i \sqcup s_i'$ *to* s_i'.

Of course, a context element s such that $s \sqsupseteq s_i \sqcup s_i'$ may not exist. In which case, the search, step 2, fails.

The meaning of this failure is that the modification should have been applied to the most abstract element in the context. This is one sense in which a modification is not at the "appropriate" level, the more general meaning concerns the amount of searching required to find s in the context.

An algorithm to search for s is:

ALGORITHM 2.

(1) *let* $j = i - 1$
(2) *if* $s_j \sqsupseteq s_i \sqcup s_i'$ *then stop* $(s = s_j)$
(3) *otherwise, let* $j = j + 1$ *and repeat the second step.*

Ideally, the check in the second step should not have to be repeated. If the modification is made at the "appropriate" level, the next highest system description should be an abstraction of both s_i and s_i'.

Consider the cost of this approach for each step. The cost of computing $s_i \sqcup s_i'$ is dependent on the ability to perform reverse engineering (which currently is not very good). The cost of the search for s is the cost of checking $s_j \sqsupseteq s_i \sqcup s_i'$ for every s_j between s and s_i in the context (an improved search would reduce this cost). With the search algorithm described above, the cost is $O(\nu(s_k, s_i \sqcup s_i'))$ where $\nu(s_k, s_i \sqcup s_i')$ is the maximum such cost. The cost for completing the context is the sum of each transformation step required (i.e. $\chi(a) + \tau(t) + \nu(a, b)$ for each specification-realization pair in the new context).

3.3.3. Cost of Downward Propagation.

Downward propagation of changes in a context provides three options in approaches. The first is redevelopment from s_i'; the second, component-wise refinement of the modified components in s_i';

and the third is use of the (integration) HPR-algorithm. The costs for each of these approaches is described below.

Redevelopment from s_i' can be carried out by the same methods used originally in development. The original context is not utilized and the cost is simply the sum of the cost of the design steps taken ($\chi(a) + \tau(t) + \nu(a, b)$ for each step taking a to b).

Component-wise refinement requires the identification of the components of s_i' which are different from those in s_i and refining them through the old context. The goal being to refine those components that differ in some way and leave the others alone except to deal with interface changes. The cost function has a similar form to that for pure redevelopment.

The component-wise refinement approach is similar to that of using the HPR-algorithm (for integration). Both approaches require the identification of changes and the combination of these with refinements. The "integration" approach would have as cost the sum of costs for each square like

$$\begin{array}{ccc} s_k & \xrightarrow{\phi_k} & s_k' \\ {\scriptstyle t_{k+1}}\downarrow & & \downarrow{\scriptstyle t_{k+1}'} \\ s_{k+1} & \xrightarrow[\phi_{k+1}]{} & s_{k+1}' \end{array}$$

from s_i' to s_n'. An estimation of this cost is that it is proportional to the computational complexity of the algorithm to find s_{k+1}'.

3.3.4. Summary.

The cost of making a change is decomposed into the sum of three costs as indicated by equation (4):

(1) making the modification from s_i to s_i',
(2) propagating the change upward, and
(3) propagating the change downward.

These costs are defined by the functions:

(1) cost of a modification: $\tau(\phi_0)$;
(2) cost of upward propagation: $O(\nu(s, s_i \sqcup s_i')) + \tau(t) + \tau(t')$, where s is the least common context element, $t : s \mapsto (s_i \sqcup s_i')$ and $t' : (s_i \sqcup s_i') \mapsto s_i'$; and
(3) cost of downward propagation (taking either the redevelopment or component-wise refinement approach): $\tau(t) + \nu(s_i', s_n') + \sum_{j=i}^{n} \chi(s_j')$, where s_n' is the implementation of s_i', and $t : s_i' \mapsto s_n'$.

4. Directions

The cost model described in the Section 3 allows the statement of an abstract expression as a solution to the modification-cost problem. This expression can be made usable by the assignment of costs, or the definition of the cost functions χ, ν, and τ. Without the ability to assign values to χ, ν and τ, the ARM remains too abstract. Further application requires a means of defining these functions.

The problem is that the ARM components are extreme abstractions of real systems engineering methodology elements (operations and productions). A useable cost model needs more specific instantiations of, or assumptions, about the ARM components. In addition, a number of factors can influence the cost. The model can be made more accurate by making assumptions that reflect the influence of these factors.

4.1. Assumptions.

For the cost model to be minimally usable, the cost functions must be defined. Defining the cost function requires the following:

(1) Making assumptions about the structure of system descriptions. In particular, the number of different structures and the means for combining them.
(2) Making assumptions about the abstraction relation on system descriptions (i.e. what structures can be refinements of others).
(3) Determining the costs of transformations.

4.1.1. System Structure.

One approach to system structure is to assume that a system is represented by (finite) directed graphs. In these graphs, nodes are system components and arcs indicate relationships between components. This is a popular approach for system representation in tools (tools for integration [Reps, 1991], and for maintenance [Yau, 1984]), and directed graphs have an easy extension into categorical constructions [Rattray and Price, 1990; Burstall and Goguen, 1977].

In general, systems are typically built from a small set of atomic components or *generators*. Regardless of the representation used, an assumption about these generators needs to be made. One possibility is to assume particular sets of generators, but a more abstract approach is to assume a finite number of types of generators. Other types of system components can then be built from the generator types (i.e. defining a type theory [Cardelli and Wegner, 1985] for systems).

4.1.2. Abstraction Relation.

Two possibilities apply to assumptions about refinement, depending on how the generators are approached. The first is to define the abstraction relation on the generators and then extend it based on the construction of larger components. The second defines refinement types using the type theory.

An example of refinement on types is refinement in the wide-spectrum language Extended ML (EML) [Sannella and Tarlecki, 1986]. In EML there are two types of system descriptions: signatures and structures. The abstraction relation is defined on EML so as to allow a system description of each type to be realized by another system description of either type. Therefore, a signature can be refined as either a signature or structure and a structure can also be refined as a signature or structure. One of these refinement types, **structure** → **signature**, is counter-intuitive.

4.1.3. Transformation.

The determination of transformation cost requires experimentation. In both approaches to defining refinement, the cost of making a refinement transformation would reflect our "experience" with either particular or classes of transformations. For example, with EML refinements, the **signature** → **signature** type refinements might be the easiest and **structure** → **signature** type refinements the hardest. In which case, the **signature** → **signature** refinement would be assigned a lower, and the **structure** → **signature** refinement, a higher cost than the other types.

The most general approach seems to be to assign transformation costs on the basis of experience with types of transformations. Having a small (discrete) number of refinement forms should make cost assignment easier, indicating the advantage of formal language systems (or formal understanding of language systems).

4.2. Additional Factors.

Making these assumptions should help make a more usable cost model. The model is only approximate due to cost factors that are not explicitly addressed by the ARM or the extensions described above. A few of these factors are:

(1) How design decisions are made.
(2) How design alternatives are identified.
(3) What type of design action is taken (redevelop, reuse, propagation).
(4) How design actions are performed.
(5) How design actions are influenced by product quality.

(6) Whether the set of principles, methods or tools employed is

   (1) fixed or variable,

   (2) complete and sound,

   (3) redundant, and

   (4) correctness "preserving".

(7) Whether the design action uses heuristics (optimizing or satisficing).

These factors affect the accuracy of the cost model, and their influence should be reflected in the cost functions.

Bibliography

Barr, M. and Wells, C. (1990), "Category Theory for Computing Science," Prentice-Hall, London.

Burstall, R. M. and Goguen, J. A. (1977), *Putting theories together to make specifications*, in "IJCAI-77," pp. 1045–1058.

Cardelli, L. and Wegner, P. (1985), *On understanding types, data abstraction, and polymorphism*, Computing Surveys **17**, 471–522.

Dasgupta, S. (1991), "Design Theory and Computer Science: Processes and Methodology of Computer Systems Design," Cambridge University Press, Cambridge, UK.

Hoare, C.A.R.; Hayes, I. J.; Jifeng, H.; Morgan, C. C.; Roscoe, A. W.; Sanders, J. W.; Sorensen, I. H.; Spivey, J. M. and Sufrin, B. A. (1987), *Laws of programming*, Communications of the ACM **30**, 672–686.

Keller, B. J. (1990), "An Algebraic Model of Software Evolution," M.S. Thesis, VPI&SU.

Keller, B. J. and Nance, R. E. (1992), *Abstraction Refinement: a model of software evolution*, Journal of Software Maintenance: Research and Practice (to appear).

Kreider, D. K. and Nance, R. E. (1991), *Objectives, principles and attributes: a structured approach to systems engineering*, Naval Surface Warfare Center Technical Digest, 22–31.

Lano, K. and Haughton, H. (1991), *A specification-based approach to maintenance*, Journal of Software Maintenance: Research and Practice **3**, 193–213.

Lehman, M. M.; Stenning, V. and Turski W. M. (1984), *Another look at software design methodology*, Software Engineering Notes **9**, 38–53.

Nance, R. E.; Keller, B. J. and Boldery, D. (1989), "Documentation Production Under Next Generation Technologies," Technical Report SRC-89-001, Systems Research Center, VPI&SU.

Neighbors, J. M. (1984), *The Draco approach to constructing software from reusable components*, IEEE Transactions on Software Engineering **SE-10**, 564–574.

Parnas, D. L. and Clements, P. C. (1985), *A rational design process: how and why to fake it*, in "Formal Methods and Software Development," vol. 2, LNCS #186, H. Ehrig, C. Floyd, M. Nivat, J. Thatcher (eds.), pp. 80–100.

Ramalingam, G. and Reps, T. (1991), *A theory of program modifications*, in "TAPSOFT '91," LNCS #494, S. Abramsky, T.S.E. Maibaum (eds.), Springer-Verlag, Berlin, pp. 137–152.

Rattray, C. and Price, D. (1990), *Sketching an evolutionary hierarchical framework for knowledge-based systems design*, in "Computer Aided Systems Theory – EUROCAST '89," LNCS #410, F. Pichler, R. Moreno-Diaz (eds.), Springer-Verlag, Berlin, pp. 360–374.

Reps, T. (1991), *Algebraic properties of program integration*, Science of Computer Programming **17**, 139–215.

Sannella, D. T. and Tarlecki, A. (1986), *Extended ML: an institution-independent framework for formal program development*, in "Proceedings of the Tutorial and Workshop on Category Theory and Computer Programming," LNCS #240, D. Pitt, S. Abramsky, A. Poigné and D. Rydeheard (eds.), Springer-Verlag, Berlin, pp. 364–389.

Yau, S. (1984), "Methodology for Software Maintenance," Final Technical Report RADC-TR83-262, NTIS AD-A143-763/1.