Morphogenesis of A Program

Giancarlo Succi, Francesco Baruchelli, Marco Ronchetti

Laboratorio di Ingegneria Informatica
Dipartimento di Informatica e Studi Aziendali
Università di Trento
via F. Zeni 8,
I-38068 Rovereto (TN)
ITALY

George Kovacs

CIM Research Laboratory
Computer and Automation Institute
Hungarian Academy of Sciences
Kende u. 13-17,
1111 Budapest
HUNGARY

Abstract: The software crisis has led to several models that describe whole or parts of the software development process. This paper introduces a model which formally describes the later stages of this cycle where the code is modified. The code is treated as a simple string without any semantic or syntactic meaning. A set of operations to manipulate strings is defined, then five operators describing the more usual changes undergone by the code are introduced. A set of properties which allows to reduce the number of transformations needed to pass from a version of the code to another one is introduced. Then the operators are extended in order to grant their invertibility and their inverse versions are defined. Finally a tool which implements the changes described by the operators, keeps trace of the transformations and allows to navigate through the versions of the code, is described.

Giancarlo Succi received his Laurea Degree in Electrical Engineering from the Università di Genova, in 1988, the M.Sc. degree in Computer Science from the State University of New York at Buffalo, in 1991, the Ph.D degree in Computer and Electrical Engineering from the Universita di Genova, in 1993, respectively. Since 1993 he has been Assistant Professor at the Universita di Trento. His research interests are in software engineering with an emphasis on software re-use.

Francesco Baruchelli received his Laurea Degree in Electrical Engineering from the Università di Padova in 1995. Since then he has been a graduate student at the Università di Trento. His research interests cover software engineering, and especially, software re-use.

George Kovacs received the B. Sc. degree in Electrical Engineering from the Technical University of Budapest, in 1966, the Ph.D degree in Electrical Engineering from the Technical University of Budapest in 1971, respectively. He was a candidate to sciences degree (CAD/CAM in electronics) from the Hungarian Academy of Sciences in 1978. As visiting researcher he spent one year in the USA, two years in the then Soviet Union, and one year in the Federal Republic of Germany. He also spent six months in Mexico as Visiting Professor. As founder of the CIM-EXP Development and Consulting Ltd., he has played a major role in the management of the company. His professional interests are in co-operative knowledge processing.

Marco Ronchetti received his Laurea Degree in Physics from the Universita di Trento in 1979. Since 1984 he has been Assistant Professor at the Università di Trento. His research interests are in software engineering, more particularly in software re-use.

1. Introduction

The current situation of crisis in the software sector has led to the definition of many models that try to formally define the software development process in order to improve it and obtain higher quality software at lower costs.

Some of these studies (such as [27], [28], [39], [42], [44] and [45]) consider the whole development process, while some others prefer to focus their attention on a single phase of the process itself. Many of these works regard the earlier steps of the software life-cycle, i.e. the analysis of the user's expectations, the definition of the product's requirements, the design of the architecture and the production of the code, but the most interesting ones in the context of this paper are the ones related to the later stages of the process during which the software resulting from the previous steps is modified.

For example, in [1] and in [26] Boyle and Muralidharan consider a series of transformations which allow to get to a FORTRAN program starting from a LISP version of the same program, passing through an intermediate version using an extended FORTRAN language allowing the use of recursion.

Freak, in [9], deals with the conversion from FORTRAN to Pascal, while Loveman, in [2], discusses a set of rules to transform a program written in an Algol-like language in order to get

an improvement in terms of speed and memory occupation.

Arsac in [3] deals with transformations on the source code, too. These changes are divided into two groups in some way complementary: a first set is constituted of transformation rules which do not alter the flow of the operations performed by the program, while the second set contains some semantic transformations which alter the behaviour of little parts of the program.

Burstall and Darlington, in [4], consider a slightly restricted set of transformations on the source code, which allows to get rid of recursion. The same problem is addressed by Irlik in [10].

The goal of a greater efficiency through program transformation is pursued by Partsch and Steinbruggen in [7], and by Partsch alone in [5], too. The same intent can be found in the work of Pettorossi and Proietti [21] with particular reference to the logic and functional languages, and in the work of Boyle and Harmer [22] who deal with the functional languages only.

Some code transformations for the maintenance of a COBOL program can be found in [6] and in [12], while the object of the work of Freak in [8] is the set of changes needed to divide existing programs into more modules.

Maintenance oriented changes are considered by Overstreet, Chen and Byrum in [15], while Merks, Dyck and Cameron consider in [13] the problem of designing a programming language which facilitates the modification of the software.

Finally some examples of transformations targeted towards the software re-use can be found in the works of Cheatham [20] and Boyle [23]. Other cases of code changes are presented in [11], [16], [17], [18], [19], [24] and [25].

Some of the models presented in these studies focus their attention only on a specific kind of transformation, for instance on the transformations performed in order to get rid of the recursion and so improving the efficiency of the program, or on the transformations that allow the translation of a program from a given programming language to a different one. The approach taken in this work is intended to model program changes in the most general way. The obtained model is useful in many situations in which there is the necessity to modify the code:

1. Changes due to the **maintenance** of the program.
2. Changes performed in order to **improve** some feature of the program.
3. Changes needed for making a chunk of code **re-usable**.

In all these cases, the only alternative to the modification of the existing code is to re-write all the program from scratch, with a natural increase of the costs.

Since modifications are an important issue in the software life-cycle, a model which describes these transformations can be useful for several reasons:

- Modelling the changes can help in better **understanding** of a program. In fact, through analyzing the changes along with the reasons which made them necessary and the effects of the changes themselves, the functions of the single parts of the program can be better understood.
- It is important to trace the changes undergone by the code. This can allow to try **to inductively infer some properties** owned by the code after a new change. If X_i is the *i*-th version of a program (the version obtained after the *i*-th change), from the knowledge of all the versions $X_1, X_2, \ldots, X_n$ and of the transformations $X_0 \rightarrow X_1$, $X_1 \rightarrow X_2, \ldots, X_{n-1} \rightarrow X_n$, along with the effects of these changes upon the code we can try to extract the elements necessary to try to understand possible effects of the next transformation $X_n \rightarrow X_{n+1}$.
- It is important **to identify the parts of the code more subjected to modifications**. This is very useful in the context of a domain analysis method.
- Such a model can be useful during the **testing** phase. In fact, if we know that a piece of code has been copied from

another program which is known to behave correctly, the testing phase can be limited to the new phase and its interaction with the copied chunk.

- The **documentation** can be produced more easily using this model. Like for the previous point, the copied parts are already documented and the related documentation can be simply copied, too.

For all these reasons we define a model to describe the changes that the source code of a program undergoes in the later stages of its life-cycle. This model is very simple, but just for its simplicity it is suited to all the situation in which a program is modified.

The code is considered as a simple string of text, without any associated syntactic or semantic meaning. Of course such an approach presents its drawbacks, first of all the fact that it cannot provide any warranty on the correctness of the code after it has been modified. On the other hand, if a syntactic or a semantic value were associated with the code, this would result in a greater complexity and would go to the detriment of the wanted generality because it would be necessary to choose a particular programming language for which to describe the changes.

The model is built of five operators which formalize five different kinds of changes that a string of characters, in our case the code, may undergo. Some properties of these operators which allow to reduce more transformations to a single one and to invert a modification to get back to a previous version of the code, have been discovered and demonstrated.

After the model has been defined, a set of functions integrated with a well- known text-editor has been realized. These functions provide the programmer with the operators constituting the model and automatically keep trace of the performed operations.

Here is the outline of the following sections.

We begin by defining some simple operations on strings on which we further base the general structure of the model (Section 2).

In Section 3 we introduce the reasons why to look for possible simplifications. Then some simplifications are introduced.

In Section 4 we describe some of the limits of the model and an extension is proposed in order to get over these limits and obtain the invertibility of the operators.

In Section 5 the functionalities and some interesting features of the tool which implements the model are described.

Finally, Section 6 provides some conclusions and directions for future work.

2. Definition of the Model

The model defined in this Section is very simple: the files with the source code are treated just like plain text without any syntactic nor semantic meaning, and the model consists of a set of operators which allows to modify these files in order to obtain new versions of the program. The way the operators act on the code is ruled by a series of parameters.

Each one of the operators that will be defined formally describes a particular transformation which is often used to modify a program for one of the reasons seen in Section 1. The operators, which allow to obtain any file starting from any other file if combined with the right parameters, are five:

- Insert (ι): the effect of this operator is the insertion of a new substring in a given position of the code.
- Delete (χ): this is the operator inverse of Insert and allows to cancel part of the code.
- Substitute (σ): this operator substitutes a determined part of the code with a new string. It can be seen as a deletion followed by an insertion.
- Replace (ρ): like Substitute, this operator performs a substitution. In this case the effect is not the substitution of a given part of the code, but the substitution of all the

occurrences of a substring passed as a parameter in the program.

- Apply (α): this operator takes part of the code and substitutes with it a variable in a function passed as a parameter. The resulting function takes then the place of the original part in the modified code.

A. Operations defined on strings

Treating the code as a simple string, there is a need for introducing some basic operations for the manipulation of strings upon which the operators will be based.

First of all a domain for these operations has to be defined. We define the alphabet Σ as the set of all the characters which can be found in the code and add to this set the void string ε not contained in the alphabet.

Then we define Σ bottom as the union between the alphabet and the void string:

$$\Sigma_{\perp} = \Sigma \cup \{\varepsilon\}$$

followed by the languages constituted by the sets of all the strings of length n:

$$\Sigma^{(1)} = \Sigma$$

$$\Sigma^{(n)} = \Sigma^{(n-1)} \times \Sigma.$$

As a convention we say that $\Sigma^{(0)} = \varepsilon$.

We can now define the language which the code will belong to: the set of all the strings of any length :

$$\Sigma^{*} = \left[\bigcup_{n=0}^{\infty} \Sigma^{(n)} \right].$$

This language Σ^{*}, along with the set of the natural numbers N, constitutes the domain and the range of the operations on strings that are now defined and of the operators of the model, too.

The operations defined on strings are 12 and can be grouped in two main categories: operations which give a string as a result of their application and operations which return an integer representing a position in the string (Table 1 reports this division).

From now on we will use the convention of identifying by number 1 the position of the first character in a string, and by number 0 the position immediately before.

- CONCAT

  This is the fundamental operation of our model; it allows to build strings starting from single characters or other strings. Its task is to accept two strings as its arguments and to return their juxtaposition the same order they have been passed. Since this function will be used many times in the course, for brevity it will be denoted by the symbol · from now on.

  The operation wants two strings as parameters and returns a string, so its domain is given by $\Sigma^{*} \times \Sigma^{*}$,

  while its range is Σ^{*}.

Table 1. Operations On Strings.

Operations returning a string	Operations returning an integer
CONCAT	LENGTH
LTEND	SAUX
LTRUNC	SEARCH
RTEND	
RTRUNC	
MID	
SUBSTRING	
LSTRING	
RSTRING	

If x and y are two strings belonging to the language Σ^{*}, the result of the concatenation of x with y will be the string obtained by juxtaposing y to x, i.e. xy. Then we have:

$$CONCAT(\cdot,\cdot):\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$$
$$CONCAT(x, y) = x \cdot y = xy$$

For example:

CONCAT("key","word")="key"·"word"="keyword"

This operation has an associative property. In fact we have:

$(x \cdot y) \cdot z = xy \cdot z = xyz$

$x \cdot (y \cdot z) = x \cdot yz = xyz$

and then:

$(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

This property allows to omit the parentheses when performing more concatenation. Then it is correct to write:

$(x \cdot y) \cdot z = x \cdot (y \cdot z) = xyz$.

- LTEND

This function takes a string as its only parameter and returns the first character of this string. If the parameter is ε, then there are no characters in the string and the void string ε itself is returned. If we identify the n-th character of a string s by the expression $s[n]$, then we can define:

$$LTEND(\cdot):\Sigma^* \rightarrow \Sigma_\perp$$
$$LTEND(\varepsilon) = \varepsilon$$
$$LTEND(s) = s[1] \quad \forall s \neq \varepsilon$$

For example:

LTEND("model")="m"

- LTRUNC

This function is in some way complementary to LTEND. It gets a string as its only parameter and truncates it returning all but the first characters. In this case the void string ε is mapped in itself, too. The resulting string is the only one that concatenated after the result of LTEND on the same original string returns the first string itself. The definition of LTRUNC is the following:

$$LTRUNC(\cdot):\Sigma^* \rightarrow \Sigma^*$$
$$LTRUNC(\varepsilon) = \varepsilon$$
$$LTRUNC(s) = w \text{ such that}$$
$$LTEND(s) \cdot w = s$$

For example:

LTRUNC("model")="odel"

- LENGTH

To be able to define the following operations, it is necessary to introduce a function which belongs to the second group of operations (the ones that return a number). This function is LENGTH and its task is to calculate the length of a string, intended as the number of characters in it. For brevity we will denote this function by the symbol $|\cdot|$, i.e. LENGTH(s)=$|s|$. The definition is recursive: the length of the void string is set equal to 0, then the length of any other string can be calculated adding one to the length of the same string without its first character. This leads to the following definition:

$$|\cdot|:\Sigma^* \rightarrow N$$
$$|\varepsilon| = 0$$
$$|s| = 1 + |LTRUNC(s)| \quad \forall s \neq \varepsilon$$

For example:

|"model"|=1+|"odel"|=1+(1+|"del"|)=...=1+(1+(1+(1+(1+|ε|))))="|)=...=1+(1+(1+(1+(1+0)))))=5

- RTEND

This operation can be considered the "mirrored" version of LTEND. In fact, it

returns the last character (the first from the right) of the string passed as the only parameter. The definition of RTEND is very similar to the one given for LTEND:

$$\mathrm{RTEND}(\cdot):\Sigma^* \to \Sigma_\perp$$
$$\mathrm{RTEND}(\varepsilon) = \varepsilon$$
$$\mathrm{RTEND}(s) = s[|s|] \quad \forall s \neq \varepsilon$$

For example:

RTEND("model")="l"

- RTRUNC

RTRUNC is related to LTRUNC in the same way RTEND is related to LTEND. It returns all but the last characters of the string passed as argument. This operation is defined as follows:

$$\mathrm{RTRUNC}(\cdot):\Sigma^* \to \Sigma^*$$
$$\mathrm{RTRUNC}(\varepsilon) = \varepsilon$$
$$\mathrm{RTRUNC}(s) = w \text{ such that}$$
$$w \cdot \mathrm{RTEND}(s) = s$$

For example:

RTRUNC("model")="mode"

- MID

All the operations defined until now allow to extract the external parts (starting from the left or from the right side) of a string. MID returns a single character in a middle position of a string. The position is specified by a number. Two parameters have to be passed to this function: the string from which to extract the character and the position of the character to be extracted. If the position falls outside the range covered by the string, i.e. if the position is equal to 0 or greater than the length of the string, the void string ε is returned. The function is defined in a recursive way, just like LENGTH. Of course if the position is 1, then the first character is returned using the previously defined LTEND, otherwise the character to be returned is found, remembering that the *n*-th character in a string is the same as the *(n-1)*-th character in the same string without its first character, which can be obtained using the operation of LTRUNC. This leads to the definition:

$$\mathrm{MID}(\cdot):\Sigma^* \to \Sigma_\perp$$
$$\mathrm{MID}(s,0) = \varepsilon$$
$$\mathrm{MID}(s,n) = \varepsilon \quad \forall n \in \mathrm{N} \text{ such that } n > |s|$$
$$\mathrm{MID}(s,1) = \mathrm{LTEND}(s)$$
$$\mathrm{MID}(s,n) = \mathrm{MID}(\mathrm{LTRUNC}(s), n-1)$$
$$\text{otherwise.}$$

For example:

MID("model",3)=MID(LTRUNC("model"), 2)=MID(LTRUNC(LTRUNC("model")),1)=

LTEND(LTRUNC(LTRUNC("model")))=LT END(LTRUNC("odel"))=LTEND("del")="d"

From this definition there follows immediately that, like for the previously defined operations:

$$\mathrm{MID}(\varepsilon,n) = \varepsilon \quad \forall n \in \mathrm{N}$$

And:

LTEND(*s*)=MID(*s*,1)

RTEND(*s*)=MID(*s*,|*s*|).

- SUBSTRING

This function returns part of a string starting from a given position and stopping to another one. We use the convention of including the character at the initial position, but not the one at the ending position: the ending position denotes the first character not to be taken. This convention results in the difference between the values of the two positions being equal to the number of characters returned by the operation, i.e. to the length of the returned string. The discrimination between the initial and the ending position is not given by the order in which the parameters are passed to

the function, but by the cardinal relation between the two numbers: the greater one denotes the final position, and the smaller one the starting position. There are some particular situations to take care of:

1. If the two numbers denoting the positions are equal, then the void string ε is returned.
2. If the initial position is equal to 0, and then it is outside the string, its value is set to 1.
3. If the final position falls outside the string, i.e. it is greater than the length of the string plus 1 ($|s|+1$, if s is the string), then it is set to $|s|+1$.

Like for other operations, this function is defined recursively. If the difference between the final and the initial position is exactly 1, then the operation can be reduced to the application of the previously defined MID, otherwise, if we suppose n and m to be the initial and the final position respectively, the result is given by the concatenation between the character at the position n, extracted using MID, and the substring starting from the position $n+1$ and ending at the position m.

We define SUBSTRING as:

$$SUBSTRING(\cdot,\cdot,\cdot): \Sigma^* \times N \times N \to \Sigma^*$$

$$SUBSTRING(s,n,n) = \varepsilon$$

$$SUBSTRING(s,n,m) = SUBSTRING(s,m,n)$$

$$\text{if } n > m$$

$$SUBSTRING(s,0,m) = SUBSTRING(s,1,m)$$

$$SUBSTRING(s,n,m) = SUBSTRING(s,n,|s|+1)$$

$$\text{if } m > |s|$$

$$SUBSTRING(s,n,n+1) = MID(s,n)$$

$$SUBSTRING(s,n,m) = MID(s,n) \cdot$$

$$SUBSTRING(s,n+1,m) \text{ otherwise.}$$

For example:

SUBSTRING("model",2,5)=MID("model",2) ·SUBSTRING("model",3,5)=

MID("model",2)·MID("model",3)· SUB

STRING("model",4,5)=

MID("model",2)·MID("model",3)·MID ("model",4)="o"·"d"·"e"="ode"

- LSTRING

This operation is a simple application of the function SUBSTRING. It returns the first n characters, where n is specified as a parameter of the given string. Its definition is the following:

$$LSTRING(\cdot,\cdot): \Sigma^* \times N \to \Sigma^*$$

$$LSTRING(s,n) = SUBSTRING(s,1,n+1)$$

For example:

LSTRING("model",3)=SUBSTRING ("model",1,4)="mod"

- RSTRING

It is another application of SUBSTRING. Here n characters are taken starting from the right end of the string. In this case the ending position is the first one after the end of the string ($|s|+1$ if s is the string), while the starting position has to be put n positions before, at $|s|+1-n$. RSTRING is defined as:

$$RSTRING(\cdot,\cdot): \Sigma^* \times N \to \Sigma^*$$

$$RSTRING(s,n) =$$

$$SUBSTRING(s,|s|-n+1,|s|+1)$$

For example:

RSTRING("model",2)=SUBSTRING ("model",4,6)="el"

- SAUX

This is the second operation we define that belongs to the group of operations which return a number as their result. It is a function

that will be used only as an auxiliary operation in defining the next operation of search of a substring in a given string (SEARCH). The purpose of SEARCH is to return the position of the first occurrence of a substring in another string starting from a given position; the function returns 0 if the substring is not found. This function will use another operation (SAUX) which is slightly different since its search starts always from the first character and the returned value when the substring is not found is not 0, but $|s|+1$, where s is the string in which the search is performed.

Two parameters are passed to SAUX: the string in which to search and the string for which to look for, in this order. There are some particular cases to take into account:

1. If the string in which the search will be performed is the void string ε, the substring will never be found and the result of the function is $|\varepsilon|+1=1$.
2. If the string searched for is the void string ε, the result will always be negative, and so it will be equal to $|s|+1$.

In any other case, the first thing to do is to check if the string to search in starts with the searched substring. If this is the case, the result of the operation is 1. If the searched substring is not found at the beginning of the given string, it could be anywhere in the middle of the string itself, so we add recursively 1 to the result of the function applied to the string without the first character (obtained using LTRUNC). The recursion will stop when the substring is found at the beginning of the remaining characters, or when there is no character left to look in and the function is applied to ε. In this case the returned value is equal to $|s|+1$ as we wanted. This leads to the definition:

$$SAUX(\cdot,\cdot):\Sigma^*\times\Sigma^*\rightarrow N$$
$$SAUX(\varepsilon,t)=1\ \ \forall t\in\Sigma^*$$
$$SAUX(s,\varepsilon)=|s|+1\ \ \forall s\in\Sigma^*$$
$$SAUX(s,t)=1 \text{ if } LSTRING(s,|t|)=t$$
$$SAUX(s,t)=1+SAUX(LTRUNC(s),t)$$
otherwise.

For example:

SAUX("A model which describes the evolution of the code","de")=

1+SAUX(LTRUNC("A model which describes the evolution of the code"),"de")=

1+(1+SAUX(LTRUNC(" model which describes the evolution of the code"),"de"))=

1+(1+(1+SAUX(LTRUNC("model which describes the evolution of the code"),"de")))=

1+(1+(1+(1+SAUX(LTRUNC("odel which describes the evolution of the code"),"de"))))=

1+(1+(1+(1+SAUX("del which describes the evolution of the code"),"de")))=

1+(1+(1+(1+1))))=5

While:

SAUX("one","I")=1+SAUX(LTRUNC("One "),"I")=1+(1+SAUX(LTRUNC("ne"),"I"))=

1+(1+(1+SAUX(LTRUNC("e"),"I")))=1+(1+(1+SAUX(ε,"I")))=1+(1+(1+1))=4

- SEARCH

The definition of this function is very simple using the auxiliary function just defined. There is one more parameter than the two seen for SAUX: the position from which to start the search. Another difference is the value returned when the search fails: it is not the length of the string plus 1, but 0. This means that the search in the void string ε will return 0 and not 1.

If the starting position is 0, the position is set to 1. In any other case, if the starting position is n, the result is calculated using SAUX: SAUX is applied to the part of the string to the right of the starting position (with the starting position included) which can be obtained using RSTRING, and n-1 is added to the result (in this way if SAUX returns 1, then the result of the addition is n, the real position of the substring). If the obtained value is smaller than or equal to the length of the string in which the search is performed, then SEARCH returns this value, otherwise the search fails, and 0 is returned.

The definition for SEARCH is the following:

$$SEARCH(\cdot,\cdot): \Sigma^* \times \Sigma^* \times N \rightarrow N$$

$$SEARCH(\varepsilon, t) = 0 \quad \forall t \in \Sigma^* \text{ and } \forall n \in N$$

$$SEARCH(s, t, 0) = SEARCH(s, t, 1) \quad \forall s, t \in \Sigma^*$$

$$SEARCH(s, t, n) = 0 \text{ if } \left\lfloor n - 1 + SAUX(RSTRING(s, |s| - n + 1), t) \right\rfloor > s$$

$$SEARCH(s, t, n) = n - 1 + SAUX(RSTRING(s, |s| - n + 1), t) \text{ otherwise.}$$

For example:

SEARCH("A model which describes the evolution of the code","de",1)=

1-1+SAUX(RSTRING("A model which describes the evolution of the code",| "A model which describes the evolution of the code"|-1+1),"de")=

0+SAUX("A model which describes the evolution of the code","de")=5

While:

SEARCH("A model which describes the evolution of the code","de",10)=

10-1+SAUX(RSTRING("A model which describes the evolution of the code",| "A model which describes the evolution of the code"|-10+1),"de")=

9+SAUX(RSTRING("A model which describes the evolution of the code",40),"de")=

9+SAUX("hich describes the evolution of the code","de")=9+6=15

And:

SEARCH("A model which describes the evolution of the code","I",1)=0

because

1-1+SAUX(RSTRING("A model which describes the evolution of the code",| "A model which describes the evolution of the code"|-1+1),"de")=50

and

50>|"A model which describes the evolution of the code"|=49

Table 2 summarizes the defined operations.

B. Definition of the operators

In this subsection we define the operators briefly introduced at the beginning of the Section.

- Insert (ι)

  The insertion of new parts is the simplest and the most common transformation that the code can undergo, and even the creation of a new file from scratch can be seen as an insertion in a void string. The parameters needed to formalize this simple operation are three: the code which is modified, the new string to insert, and the position of the insertion. The position of the insertion indicates the position which will be held by the first character of the inserted string; this means that it has to be equal to $|s|+1$ if a new text has to be added at the end of the code s.

Table 2. Operations On Strings: Domain, Range and Description

Operation	Domain	Range	Description
CONCAT(·,·)	$\Sigma^* \times \Sigma^*$	Σ^*	Concatenates two strings
LTEND(·)	Σ^*	$\Sigma_\perp$	First character of a string

LTRUNC(·,·)	Σ^*	Σ^*	String without the first character
$\lvert\cdot\rvert$ (LENGTH)	Σ^*	N	Length of a string
RTEND(·)	Σ^*	$\Sigma_\perp$	Last character of a string
RTRUNC(·)	Σ^*	Σ^*	String without the last character
MID(·,·)	$\Sigma^* \times N$	$\Sigma_\perp$	A character in the middle of a string
SUBSTRING(·,·,·)	$\Sigma^* \times N \times N$	Σ^*	Extracts a substring
LSTRING(·,·)	$\Sigma^* \times N$	Σ^*	Substring starting from the left
RSTRING(·,·)	$\Sigma^* \times N$	Σ^*	Substring starting from the right
SAUX(·,·)	$\Sigma^* \times \Sigma^*$	N	Auxiliary search function
SEARCH(·,·,·)	$\Sigma^* \times \Sigma^* \times N$	N	Search a substring in a string

When the position of the insertion falls outside the range covered by the code, the string is inserted at its beginning or at its end.

In every case, the insertion can be done by concatenating the substring preceding the position of insertion (obtained with LSTRING) with the new string and then with the substring following the position of insertion (obtained using RSTRING). This leads to:

$$\iota(\cdot,\cdot,\cdot): \Sigma^* \times N \times \Sigma^* \rightarrow \Sigma^*$$
$$\iota(c,0,s) = \iota(c,1,s) \quad \forall c,s \in \Sigma^*$$
$$\iota(c,n,s) = \iota(c,|c|+1,s) \quad \forall c,s \in \Sigma^* \text{ and } \forall n \in N \text{ such that } n > |c|+1$$
$$\iota(c,n,s) = LSTRING(c,n-1) \cdot s \cdot RSTRING(c,|c|-n+1) \text{ otherwise.}$$

For example:

ι("This model is simple","very ",15)="This model is very simple"

- Delete (χ)

This operator formalizes the change inverse to the insertion: the deletion of part of the code. For this operation it is necessary to specify with two parameters the part of the code to delete; these parameters are the position of the first character to cancel and the position following the last character to cancel. The initial and the final positions are identified by their cardinal order and not by the order in which they are passed to the operator. Whenever one of the parameters falls outside the range covered by the code, it is set to the boundary value.

The transformation is simply obtained by concatenating the characters preceding the deleted part with the ones following the same part. Formally:

$$\chi(\cdot,\cdot,\cdot): \Sigma^* \times N \times N \rightarrow \Sigma^*$$
$$\chi(c,b,e) = \chi(c,e,b) \text{ if } b > e$$
$$\chi(c,0,e) = \chi(c,1,e) \quad \forall c \in \Sigma^*$$
$$\chi(c,b,e) = \chi(c,b,|c|+1) \text{ if } e > |c|+1$$
$$\chi(c,b,e) = LSTRING(c,b-1) \cdot RSTRING(c,|c|-e+1) \text{ otherwise.}$$

For example:

χ("This model is very simple",15,20)="This model is simple"

- Substitute (σ)

Another common kind of transformation is the substitution of a chunk of code with another one. The substitution is performed by deleting part of the code and then inserting the new string in the position previously occupied by the cancelled characters. This can be obtained by the application of the two operators already defined: Delete and Insert. The required parameters, apart from the code to modify, consist in the new string to insert and the positions of the beginning and the end of the part to substitute (as for Delete, the positions

are identified by their cardinal order and not by the order in which they are passed to the operator). The definition is based on the use of χ and ι, this means that the particular cases are managed by these operators and there is no need to specify many cases. The definition is simply:

$$\sigma(\cdot,\cdot,\cdot,\cdot): \Sigma^* \times N \times N \times \Sigma^* \rightarrow \Sigma^*$$
$$\sigma(c,b,e,s) = \sigma(c,e,b,s) \quad \forall c,s \in \Sigma^*$$
$$\text{if } b > e$$
$$\sigma(c,b,e,s) = \iota(\chi(c,b,e),b,s) \text{ otherwise.}$$

For example:

σ("This model is rather simple",15,21,"very")="This model is very simple"

Replace (ρ)

As said at the beginning of this Section, this operator looks for all the occurrences of a given substring in the code and substitutes them with a new string. The parameters to pass to the operator are the code to modify, the searched substring and the string to insert. If the searched substring is not present in the

ρ("for(int num1=0;num1<5;num1++)
if (check==num1)
break;","num1","num2")=
LSTRING("for(int num1=0;num1<5;num1++)
if (check==num1)
break;",
SEARCH("for(int num1=0;num1<5;num1++)
if (check==num1)
break;","num1",0)-1)·"num2"·
ρ(RSTRING("for(int num1=0;num1<5;num1++)
if (check==num1)
break;"," for(int num1=0;num1<5;num1++)
if (check==num1)
break;"|-
SEARCH("for(int num1=0;num1<5;num1++)

code, the result of the transformation is the unmodified code. When an occurrence of the searched string is found, the replacement can be done as seen for Substitute by concatenating the preceding part of the string with the new string and the part of the code following the last character of the searched string. This is not sufficient, because all the other occurrences of the string have to be found, too. For this reason the function is defined recursively and the operator is applied to the string following the found string before its concatenation with the left part of the string and the new string to insert. This yields the following definition:

$$\rho(\cdot,\cdot,\cdot): \Sigma^* \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$$
$$\rho(s,t,w) = s \text{ if } SEARCH(s,t,0) = 0$$
$$\rho(s,t,w) = LSTRING$$
$$(s, SEARCH(s,t,0) - 1) \cdot w \cdot$$
$$\rho(RSTRING(s, |s| - SEARCH(s,t,0) +$$
$$1 - |t|), t, w) \text{ otherwise.}$$

For example:

```
                    if (check==num1)
                      break;","num1",0)+1-|"num1"|),"num1","num2")=
"for(int "·"num2"·ρ("=0;num1<5;num1++)
                  if (check==num1)
                    break;","num1","num2")=...=
"for(int num2=0;num2<5;num2++)
      if (check==num2)
        break;"
```

- Apply (α)

  This transformation is not as simple as the previous one. It consists of the "application" of a function written in the programming language used in the code as part of the code itself. For example, if a sequence of instructions has to be repeated, a certain number of times, this sequence can be seen as the argument of a for-loop. The parameters to pass to the operator are: the code to modify, the starting and ending position of the chunk of code to take as argument of the function to apply, the function to apply, and the string that is the variable in the function to apply and which will be substituted by the selected chunk of code. If the substring identifying the variable is not present in the function, the code is returned by the operator without any change, otherwise the transformation consists of a substitution (performed by the operator Substitute) of the selected part of the code with the function in which all the variable occurrences have been replaced (by the operator Replace) by the selected chunk of code. The definition of this operator is rather simple:

$$\alpha(\cdot,\cdot,\cdot,\cdot,\cdot): \Sigma^* \times N \times N \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$$
$$\alpha(c,b,e,f,t) = c \text{ if } \rho(f,t, SUBSTRING(c,b,e)) = f$$
$$\alpha(c,b,e,f,t) = \sigma(c,b,e,\rho(f,t, SUBSTRING(c,b,e))) \text{ otherwise.}$$

For example:

```
α("int i=0;
   i++;
   exit(-2);",10,13,"if (x1<10)
                      a=x1;
                    else
                      exit(-1)","x1")=
σ("int i=0;
   i++;
   exit(-2);",10,13,ρ( "if (x1<10)
                        a=x1;
                      else
                                                    exit(-
1)","x1",SUBSTRING("int i=0;

i++;

exit(-2);",10,13)))=
σ("int i=0;
   i++;
   exit(-2);",10,13,"if (i++<10)
                      a=i++;
                    else
                      exit(-1)")=
"int i=0;
 if (i++<10)
   a=i++;
 else
   exit(-1);
 exit(-2);"
```

The operators that constitute the model are summarized in Table 3.

Table 3. The Operators of the Model

Operator	Domain	Range	Description
$\iota(\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times \Sigma^*$	Σ^*	Insertion
$\chi(\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times N$	Σ^*	Deletion
$\sigma(\cdot,\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times N \times \Sigma^*$	Σ^*	Substitution
$\rho(\cdot,\cdot,\cdot)$	$\Sigma^* \times \Sigma^* \times \Sigma^*$	Σ^*	Replacement of a substring
$\alpha(\cdot,\cdot,\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times N \times \Sigma^* \times \Sigma^*$	Σ^*	Application of a function

3. Properties of the Operators

This Section deals with some simplifications possible when more operators, equal or different, are applied consecutively to the same piece of code.

This search for simplifications in the use of the operators in order to reduce the number of transformations necessary for getting from one version to another of the code could seem to go against the purpose of the model of describing and recording every single change in the code. This is not true, because the introduction of a series of rules to simplify the sequence of the modifications can be useful when trying to verify some property of the code. In fact, a small number of changes interesting for different zones of the code may be more meaningful for a long series of transformations always regarding the same chunk of code because in this case it is more difficult to see the ultimate goal of these microchanges. These considerations apply when we want to try to understand which features of the original program can be found in the modified version, which ones are lost in the transformations, and which are the newly acquired ones.

For example, let us suppose A to be a chunk of code with the set of related properties $P(A)$, and $T(A)$ to result from the application of a series s of n operators to A, and let us try to understand which is the set of properties $P(T(A))$ of the resulting program. It is intuitive that if there exists a sequence of transformations s' of length m, where it is $m<n$ which brings from A to $T(A)$, then it will probably be easier to infer the properties of $T(A)$ starting from s' than starting from s. A deletion followed by an insertion in the same position can be seen as a substitution, and this point of view can be more meaningful when trying to understand the effects of the transformation.

- The simplifications introduced in the rest of this Section are divided into two groups:
  - simplifications possible when the same operator is applied twice consecutively
  - simplifications possible when two different operators are applied consecutively.

It must be noted that all the properties are valid only when the parameters identifying the positions in the strings are in the range covered by the strings which they are related to and when the positions are passed to the operators with the beginning value preceding the ending one.

A. Simplifications involving a single operator

All the simplifications introduced in this Subsection refer to two consecutive applications of the same operator. In many cases these are properties which reduce to one the number of transformations used to get from a version to another of the code.

The simplifications do not apply each time an operator is applied twice consecutively, their application is only possible when the parameters satisfy some condition.

An example of the way these properties can be demonstrated is to be found in Appendix.

- Insert||Insert[1]
- If certain conditions are verified, it is easy to see how two applications of Insert can be reduced to a single application of the same operator to the code, while the other application of ι interests one of the inserted strings. This happens when the inserted strings are in some way adjacent in the resulting code. There are three possible cases in which this condition is verified:

    1. With the result of the double transformation the second inserted string is immediately followed by the first one. This happens when the second insertion is performed with the same insertion point of the first transformation, and can be seen as the insertion of the second string concatenated with the first.
    2. This case is the opposite of the previous one: the second string inserted appears immediately after the first one. This happens when the position of the second insertion is equal to the position of the first insertion plus the length of the first inserted string, and comes to the same result of the insertion of the first string concatenated with the second.
    3. The last case in which a double insertion can be simplified occurs when in the resulting code the second string appears in the middle of the first. This happens when the second insertion point assumes a value included between the first insertion point and the first insertion point plus the length of the inserted string. In this case the transformations can be reduced to the insertion of a string obtained by concatenating the left part of the first string with the second string and the right part of the first string once again. This concatenation is indeed an insertion, so we have a double insertion again, but this time the first insertion is made on the inserted string and not on the code. This simplification does not reduce the number of operations that have to be performed, but can be useful when other operators are then applied to the code.

It has to be noted that the first two cases are only particular cases of the third one, where the first or the last substrings concatenated are equal to the void string ε. The property can be written as follows:

$$\iota(\iota(c,n,s),m,t) = \iota(c,n,\iota(s,m-n+1,t)) \text{ if } n \le m \le n+|s|$$

- Delete‖Delete

    Two consecutive applications of Delete can be simplified when the position of the first character removed by the first cancellation falls above one of the characters which belong to the part deleted by the second transformation. This happens in two situations, of which the first is a particular case of the second:

    1. The two initial positions of the chunk to delete are equal. In this case the transformation can be reduced to the cancellation of a number of characters equal to the sum of the lengths of the two chunks deleted starting with the character at the initial position for both operations.
    2. The same kind of simplification can be performed when the initial position of the second transformation falls before the initial position of the first deletion, and the ending position falls after the same position. In this

[1]From now on the combination of two operators will be denoted by the symbol ‖. For example, if ω_1 and ω_2 are two operators, $\omega_1 \square \omega_2$ indicates the operator obtained from their combination if ω_2 is applied before ω_1. This means that $\omega_1 \| \omega_2(s) = \omega_1(\omega_2(s))$.

case the number of characters removed is still equal to the sum of the lengths of the two strings deleted, but the initial position is the initial position of the second transformation.

This simplification can be expressed formally as:

$$\chi(\chi(c,b,e),m,n) = \chi(c,m,n+e-b) \quad \text{if } m \le b \le n$$

- Substitute‖Substitute

  The different cases in which a double substitution can be simplified amount to four. In all these situations the chunk substituted by the second operator contains part of the string inserted in the first transformation, or, at least, is adjacent to this string.

  1. The simplest situation occurs when the whole string inserted in the first transformation is fully contained in the chunk deleted by the second Substitute, i.e. if the initial and the ending positions of the second substitution fall respectively before and after the first and the last characters of the inserted string. In this case the transformations can be reduced to a simple substitution of the chunk made up of a number of characters equal to the sum of characters deleted by the two original transformations and starting from the initial position used in the second application of the operator. The inserted string is the one inserted by the second Substitute.

  2. The second situation considered occurs when the beginning position of the substring to remove falls to the left (or on) the first character of the string inserted by the first Substitute, but the ending position falls in the middle of the same string. The transformations cannot be reduced to the application of a single operator, but one of the operators can change: the resulting transformations are a deletion followed by a substitution. In fact, the result is the same as that which can be obtained by substituting all the characters starting from the initial position given to the second Substitute, for a total of characters equal to the characters removed by the first operation plus the characters belonging to the original code deleted by the second one, by the concatenation of the second string inserted with the first string inserted, where the characters deleted by the second transformation have been removed by an application of Delete. In this case the number of operators is not reduced, but the simplification can be useful because the new sequence of changes could allow to perform other simplifications that were not possible before this manipulation.

  3. This situation is complementary to the second one. It occurs when the initial position of the second chunk to delete falls in the middle of the string inserted by the first Substitute, and the ending position falls to the right of the last character of the same string. The result is equal to the one obtained with a deletion followed by a substitution, just like for the previous case, with a change in the parameters used, to be seen in the formalization of the property.

  4. The last situation is complementary to the first one and occurs when all the characters removed belong to the string inserted by the first application of Substitute. This means that the starting position of the chunk deleted by the second transformation falls to the left of or above the inserted string, while the ending position falls to the right of or above the last character of the same string. The transformations can then be reduced to a simple substitution of the characters deleted by the first application of Substitute with a string obtained by substituting the parts removed by the second application of σ from the first inserted string with

the string inserted by the second change.

All these simplifications can be written as follows:

$$\sigma(\sigma(c,b,e,s),m,n,t) =$$
$$\sigma(c,m,n-|s|+e-b,t) \text{ if } m \le b \text{ and } n \ge b+|s|$$
$$\sigma(c,m,e,t\cdot\chi(s,1,n-b+1)) \text{ if } m \le b \text{ and } b \le n \le b+|s|$$
$$\sigma(c,b,n-|s|+e-b,\chi(s,m-b+1,|s|+1)\cdot t)$$
$$\text{if } n \ge b+|s| \text{ and } b \le m \le b+|s|$$
$$\sigma(c,b,e,\sigma(s,m-b+1,n-b+1,t))$$
$$\text{if } m \ge b \text{ and } n \le b+|s|$$

- Replace‖Replace

  For this combination of transformations, the found property does not consist of a real simplification, but of a simple permutation of the parameters. This property will be useful in the definition of some of the properties that will be introduced in the next Subsection.

  The property can be applied when all the occurrences of a given string s are substituted with another string t, and then all the occurrences of t are replaced by another string u. This sequence of transformations can be deceptive. In fact, one could think that the same result could be obtained simply if replacing all the occurrences of s by the string u, but this is not true, since this way the occurrences of t present in the original code would not be replaced by the string u, as it happens if the sequence of the two Replace is applied. The only possible simplification consists in changing the order of the replacement. The same result can be obtained, in fact, by replacing all the t by s, and then all the s by u, or , by replacing all the t by u, and then all the s by u, or, finally, by replacing all the s by u, and then all the t by u. This means that:

  $$\rho(\rho(c,s,t),t,u) = \rho(\rho(c,t,s),s,u) = \rho(\rho(c,t,u),s,u) = \rho(\rho(c,s,u),t,u)$$

  While it is not always true that:

  $$\rho(\rho(c,s,t),t,u) = \rho(c,s,u)$$

B. Simplifications involving two different operators

The possible combinations of two operators taken from a set of five are twenty-five. Not all these combinations determine some simplification. Table 4 summarizes the situation with a ♦ denoting the combinations for which some properties have been found. In this Table the first applied operator can be found on the horizontal axis, while the vertical axis refers to the second operator.

Table 4.
Simplifications

	ι	χ	σ	ρ	α
ι	♦	♦	♦		
χ	♦	♦	♦		
σ	♦	♦	♦		
ρ				♦	
α	♦		♦	♦	

- Insert‖Delete

  When a cancellation is followed by an insertion, a simplification is possible only if the starting position of the deleted chunk coincides with the insertion point. In this case, the result is the same to that which can be obtained through a simple substitution:

  $$\iota(\chi(c,b,e),b,s) = \sigma(c,b,e,s)$$

- Delete‖Insert

  This combination of operators is not as simple as the previous one. A simplification, or an interchange in the order of the operators, can be performed when one or more of the deleted characters belong to the string inserted by ι, or at least the removed chunk and the inserted string are adjacent. There

are four possible situations to take care of:

1. The deleted chunk contains the whole string inserted by ι. This happens when the starting position (b) of the part of code to remove falls to the right of or coincides with the insertion point (n), and the ending position of the part to delete (e) falls to the right of the last character of the inserted string (s), so it is greater than or equal to $n+|s|$. In this situation, after this double transformation there will be no trace of the inserted string and the resulting code can be obtained by a single cancellation.

2. The second situation considered occurs when the starting position is still to the left of the insertion point, like in the previous case, but the ending position falls in the middle of the inserted string, i.e. to the left of the last character of this string, but to the right of the insertion point. In this case it is not possible to simplify the transformation using a single operator, but we can invert the order of the modifications if we add another change not to be performed on the code, but on the string which is inserted by ι. In fact we can first delete the part of the original code that is removed by χ and then insert the part of the new string which is not of interest cancellation (obtained with another application of χ).

3. This situation is complementary to the previous one: it occurs when the ending position of the chunk to delete falls to the right of the last character of the inserted string, while the starting position of the same chunk falls in the middle of this string. In this case the order of the operators can be changed adding another transformation, just like in the previous situation.

4. The last considered situation occurs when all the deleted characters belong to the inserted string. In this case the result of the double change can be obtained by deleting the removed characters from the originally inserted string and then inserting the result in the code.

All these considerations can be formalized as follows:

$$\chi(\iota(c,n,s),b,e) =$$
$$\chi(c,b,e-|s|) \text{ if } b \le n \text{ and } e \ge n+|s|$$
$$\iota(\chi(c,b,n),b,\chi(s,e-n+1)) \text{ if } b \le n \text{ and } n \le e \le n+|s|$$
$$\iota(\chi(c,n,e-|s|),n,\chi(s,b-n+1,|s|+1)) \text{ if } n \le b \le n+|s| \text{ and } e \ge n+|s|$$
$$\iota(c,n,\chi(s,b-n+1,e-n+1)) \text{ if } n \le b \le n+|s| \text{ and } n \le e \le n+|s|$$

- Substitute∏Insert

The different situations under which some kind of property can be found for this combination of operators, are analogous to the ones remarked for the previous combination:

1. If the whole string inserted by ι is contained in the chunk that is removed by σ, after the substitution there is no trace left of this string. This means that the same result can be obtained through a simple substitution interesting for the characters of the original code that are removed by the operator.

2. When the starting position of the substituted chunk falls to the left of or above the first character of the inserted string and the ending position of this chunk falls in the middle of the same string, the two transformations can be inverted in their order if we add a deletion on the string inserted by ι. In fact, we can first substitute the characters belonging to the original code, and then insert at the end of the string added by σ the characters of the

string added by the original ι not interesting for σ (extracted using χ).

3. This situation is complementary to the previous one. In this case the ending position of the deletion falls to the right of the last character of the inserted string, while the starting position falls in the middle of this string. The order of the operators can be inverted just like in the previous situation.

4. When the substitution refers only characters belonging to the inserted string, we can get the same result if we first substitute these characters and then insert in the code the result of this transformation.

This means that:

$$\begin{array}{l}\sigma(\iota(c,n,s),b,e,t) = \\ \sigma(c,b,e-|s|,t) \text{ if } b \le n \text{ and } e \ge n+|s| \\ \iota(\sigma(c,b,n,t),b+|t|,\chi(s,1,e-n+1)) \\ \text{ if } b \le n \text{ and } n \le e \le n+|s| \\ \iota(\sigma(c,n,e-|s|,t),n,\chi(s,b-n+1,|s|+1)) \\ \text{ if } n \le b \le n+|s| \text{ and } e \ge n+|s| \\ \iota(c,n,\sigma(s,b-n+1,e-n+1,t)) \\ \text{ if } n \le b \le n+|s| \text{ and } n \le e \le n+|s|\end{array}$$

- Insert‖Substitute

  The order of these two transformations can be inverted when the insertion point of ι coincides with or is immediately adjacent to the position of one of the characters inserted by σ. In this case, the same result can be obtained if we substitute the chunk of code deleted by σ with the string obtained through the insertion of the characters passed to ι in the string passed to σ:

$$\iota(\sigma(c,b,e,s),n,t) = \sigma(c,b,e,\iota(s,n-b+1,t)) \text{ if } b \le n \le b+|s|$$

- Substitute‖Delete

  This combination allows a simplification when the substituted chunk begins before the position occupied by the string removed by Delete, and ends after the same position. In this situation, the whole deletion of all the characters can be made by the single Substitute:

$$\sigma(\chi(c,b,e),m,n,s) = \sigma(c,m,n+e-b,s) \text{ if } m \le b \le n$$

- Delete‖Substitute

  Once again The situations in which a simplification is possible for this combination of operators are four. In all these cases the chunk deleted by χ contains part of the string inserted by σ or is at least adjacent to it.

  1. The simplest situation occurs when all the characters inserted by σ in substitution of a chunk of the code, are contained in the part of the code deleted by χ. In this case it is evident that the transformation is reduced to a simple deletion of the characters deleted by χ and present in the original code together with the characters removed by σ to make room for the inserted string.

  2. When all the characters deleted belong to the string inserted by σ, we can only invert the order in which the operators are applied: the modified code can be obtained if removing the characters deleted by χ from the string inserted by σ, and then applying σ with the result of this application of Delete passed as the string to insert in the code.

  3. This situation and the following one consider the cases where the characters removed by Delete belong part to the original code and part to the string inserted by Substitute. If the starting position of the chunk to delete falls to the left of the first character belonging to the string inserted by σ, and the ending position of the same chunk falls in the middle of the same string, the transformations produce the same result of a substitution of the chunk containing all the characters removed by σ plus the characters deleted by χ

and belonging to the original code with a string obtained by deleting the characters removed by χ from the string inserted by σ.

4. When the starting position of the chunk to delete falls in the middle of the string inserted by σ and the ending position falls to its right, the same inversion in the order of the operators as that introduced in case 3, can be applied. The only distinction between these two situations consists of a different value for some of the parameters.

The simplifications valid for this combination are the following:

$$\chi(\sigma(c,b,e,s),m,n) =$$
$$\sigma(c,m,n-|s|+e-b) \text{ if } m \le b \text{ and } n \ge b+|s|$$
$$\sigma(c,b,e,\chi(s,m-b+1,n-b+1)) \text{ if } b \le m \le b+|s| \text{ and } b \le n \le b+|s|$$
$$\sigma(c,m,e,\chi(s,1,n-b+1)) \text{ if } m \le b \text{ and } b \le n \le b+|s|$$
$$\sigma(c,b,e+n-b-|s|,\chi(s,m-b+1,|s|+1)) \text{ if } b \le m \le b+|s| \text{ and } n \ge b+|s|$$

- Apply∏Insert

The only simplification possible for this combination of operators occurs when the starting and the ending positions of the chunk of code that will become argument of the function passed to Apply coincide with the first and the last characters of the string inserted by ι. In this case the double transformation can be seen as the insertion of a string obtained by replacing all the occurrences of the variable in the function by the string inserted by ι:

$$\alpha(\iota(c,n,s),n,n+|s|,f,t) = \iota(c,n,\rho(f,t,s))$$

- Apply∏Substitute

This property is very similar to the previous one: it can be applied only when the starting and the ending positions of the chunk of code that will become argument of the function passed to Apply coincide with the first and the last characters of the string inserted by σ.

In this case, to get the same result in another way, the first operator to apply is Replace as in the previous property, but the second one is Substitute instead of Insert:

$$\alpha(\sigma(c,b,e,s),b,b+|s|,f,t) = \sigma(c,b,e,\rho(f,t,s))$$

- Apply∏Replace

The last property we introduce regards the combination between Apply and Replace. Particular care has to be taken in this case, because Replace can induce some errors as we have seen for the combination of this operator with itself. If we consider an application of α where u is the string chosen to represent the variable in the function f that will be applied to a chunk of the code by the operator, and we substitute all the occurrences of a string t in the function f with u, we could think that the result of the transformations is the same as that we can simply obtain if applying α with the original function f where the variable is denoted by t and no more by u. This is false. In fact, proceeding like this the occurrences of u that are present in the original version of f would not be considered as variable as is done by the double transformation which we started from. The only property valid for this combination of operators is a simple permutation of the parameters t and u:

$$\alpha(c,b,e,\rho(f,t,u),u) = \alpha(c,b,e,\rho(f,u,t),t)$$

4. Extension of the Model

A major limit of the model introduced in the previous Section is the non-invertibility of the operators. It is not possible to rebuild the version of the code preceding a transformation starting from the knowledge of the resulting code and the kind of operator applied. For example, after an application of Delete it is not enough to know the position of the removed characters and the resulting code to get the original version of the code because no data give information about the cancelled part. This situation can be a limit when

we keep trace of the transformations undergone by a program, since every time we have to get a desired previous version of the code it is necessary to start from the very first version and apply all the subsequent transformation until the version is reached.

For this reason in the rest of this Section we will introduce an extended version of the operators previously defined, and then the inverse operators for four of the five transformations (the inversion of Replace would require the introduction of more complex ranges. An approximation not formally demonstrated of this inversion has been introduced in the tool described in Section 5).

A. Extended operators

The extension of the operators strongly depends on the operators introduced in the previous Section, since the operation performed on the code keeps unchange. In the following the extended operators will be denoted by a star (For example ι^* is the extended version of Insert).

- Insert (ι^*)

  The inverse of an operator must return the parameters passed to the operator starting from its results, this means that in the case of Insert the result of its inverse must contain the unmodified code, the insertion point and the inserted string. The extended version of Insert must return the modified version of the code plus some data which to extract the insertion point and the inserted string from. The insertion point is returned as it is by ι^*, while the inserted string could be drawn out from three different data: the position of its last character, its length, or the string itself.

  The chosen solution is the first for two reasons: a number can be stored in less space than a string, and this can be useful when keeping track of a lot of transformations. Returning a number which identifies a position means that the range of Insert coincides with the domain of Delete.

  The extended Insert returns three elements in this order:

  1. The modified code.
  2. The insertion position.
  3. The position following the last character of the inserted string.

  The formal definition of ι^* is the following:

$$\iota^*(\cdot,\cdot,\cdot):\Sigma^* \times \mathrm{N} \times \Sigma^* \rightarrow \Sigma^* \times \mathrm{N} \times \mathrm{N}$$

$$\iota^*(c,n,s) = (\iota(c,n,s), n, n+|s|)$$

- Delete (χ^*)

  The inverse of Delete must return the code before the transformation and the starting and the ending positions of the deleted chunk. In order to rebuild the unmodified code it is necessary to know the deleted string and its position in the original version of the code. In this case we cannot choose among more solutions, however a symmetry occurs since the range of χ^* coincides with the domain of ι^*. The deleted string cannot be extracted directly from one of the parameters, but it is obtained after applying the operation SUBSTRING:

$$\chi^*(\cdot,\cdot,\cdot):\Sigma^* \times \mathrm{N} \times \mathrm{N} \rightarrow \Sigma^* \times \mathrm{N} \times \Sigma^*$$

$$\chi^*(c,b,e) = (\chi(c,b,e), \min(b,e), \mathrm{SUBSTRING}(c,b,e))$$

- Substitute (σ^*)

  The parameters required by σ are four: the code to modify, the starting and the ending positions of the chunk of the code to substitute and the new string to insert. These are the data that its inverse must return. Since this operator is constituted by a combination of Insert and Delete, its extended version is similar to both of extended versions of these operators.

In fact it returns the modified code and the starting position of the inserted string (like in both ι^* and χ^*), the ending position of the inserted string (like in ι^*) and the removed characters (like in χ^*). There is still a symmetry in the domain and the range, since they coincide. In fact we define:

$$\sigma^*(\cdot,\cdot,\cdot,\cdot): \Sigma^* \times N \times N \times \Sigma^* \rightarrow$$
$$\rightarrow \Sigma^* \times N \times N \times \Sigma^*$$
$$\sigma^*(c,b,e,s) = (\sigma(c,b,e,s), \min(b,e), \max(b,e), \text{SUBSTRING}(c,b,e))$$

- Apply (α^*)

The five parameters that have to be rebuilt starting from the result of α^* are the ones passed to the operator: the code before the transformation, the starting and the ending positions of the part of the code that becomes argument of the applied function, the function and the string denoting the variable in the function. This transformation is performed with a substitution, that means that to get back we need to know the position occupied by the chunk removed and the chunk itself, plus the ending position of the inserted string (in this case the function applied to the substituted part of the code). The function and the variable are returned directly; this happens because of the difficulty met with in trying to invert Replace. The complete definition of α^* is the following:

$$\alpha^*(\cdot,\cdot,\cdot,\cdot,\cdot): \Sigma^* \times N \times N \times \Sigma^* \times \Sigma^* \rightarrow$$
$$\rightarrow \Sigma^* \times N \times \Sigma^* \times \Sigma^* \times \Sigma^*$$
$$\alpha^*(c,b,e,f,t) = (\alpha(c,b,e,f,t), b, \text{SUBSTRING}(c,b,e), f, t)$$

Table 5 summarizes the extended operators.

Table 5. The Extended Operators

Operator	Domain	Range
$\iota^*(\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times \Sigma^*$	$\Sigma^* \times N \times N$
$\chi^*(\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times N$	$\Sigma^* \times N \times \Sigma^*$
$\sigma^*(\cdot,\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times N \times \Sigma^*$	$\Sigma^* \times N \times N \times \Sigma^*$
$\alpha^*(\cdot,\cdot,\cdot,\cdot,\cdot)$	$\Sigma^* \times N \times N \times \Sigma^* \times \Sigma^*$	$\Sigma^* \times N \times \Sigma^* \times \Sigma^* \times \Sigma^*$

B. Inverse operators

After the introduction of an extended version for the operators, it is now possible to define the inverse for these transformations. The missing inversion of Replace is due to its recursive nature. It is the operator itself which looks for the occurrences of the string to replace, and the number of the substitutions is not fixed. This would require a range of variable dimensions for an extended operator which allows to rebuild the original code. It is easy to note that a simple exchange between the replaced and the replacing strings is not a correct one. In fact, if we assume to replace all the occurrences of string s by string t, the replacement of all the occurrences of t in the resulting code by s could not give the original code as its result, since this way the t originally present in the code would have been substituted with s, too.

It has to be noted that the inversion of Delete and Sigma returns the starting position of a chunk of code always before its ending position, even if this were not the case when the transformation had been applied.

The demonstration of the fact that the inverse operator applied on the result of the direct operator returns the original parameters can be carried on the same way as followed when

demonstrating the validity of the simplification, an example of which can be found in Appendix.

- Insert

  Of course the domain and the range of the inverse operator must coincide with the range and the domain of the direct operator, respectively. This is true for each of the inverse operators introduced in the following.

  The original code can simply be obtained by deleting the characters inserted through direct transformation. The positions of the first and the last one of these characters are the second and the third elements returned by the direct operator. The insertion point coincides with the second element of the result, while the inserted string can easily be obtained using SUBSTRING:

$$(\iota^*)^{-1}(\cdot,\cdot,\cdot): \Sigma^* \times N \times N \to \Sigma^* \times N \times \Sigma^*$$
$$(\iota^*)^{-1}(c,b,e) = (\chi(c,b,e), b, \text{SUBSTRING}(c,b,e))$$

It is interesting to note that this definition is exactly the same as that given to the extended version of Delete. This justifies the choice made for the third element returned by ι^*.

- Delete

  The inversion of Delete is somehow symmetrical to the one performed for Insert. In this case in order to rebuild the original version of the code we have to reinsert the characters removed by the transformation. The other parameters are immediately drawn out from the result of χ^*:

$$(\chi^*)^{-1}(\cdot,\cdot,\cdot): \Sigma^* \times N \times \Sigma^* \to \Sigma^* \times N \times N$$
$$(\chi^*)^{-1}(c,n,s) = (\iota(c,n,s), n, n+|s|)$$

  If we consider the definition of the extended version of Insert we can note that this is the same as the one just introduced for the inverse of Delete. This along with the symmetrical consideration made after the definition of $(\iota^*)^{-1}$, means that Insert and Delete are one the inverse of the other not only intuitively, but from a formal point of view, too.

- Substitute

  To invert a substitution we have simply to substitute the new characters for the old ones. This leads to the following definition:

$$(\sigma^*)^{-1}(\cdot,\cdot,\cdot,\cdot): \Sigma^* \times N \times N \times \Sigma^* \to$$
$$\to \Sigma^* \times N \times N \times \Sigma^*$$
$$(\sigma^*)^{-1}(c,i,f,w) = (\sigma(c,i,f,w), i, i+|w|, \text{SUBSTRING}(c,i,f))$$

This is exactly the same definition given to the direct version of the operators. That means that Substitute is the inverse of itself.

- Apply

  Some of the parameters passed to Apply , having to be returned by $(\alpha^*)^{-1}$ are returned directly in the result of α^*. These parameters are the function and its variable. The original version of the code is rebuilt with a substitution (since the core of Apply itself is a substitution and the inverse of a substitution is again a substitution), while the other parameters can easily be extracted from the other elements returned by α^* as follows:

$$(\alpha^*)^{-1}(\cdot,\cdot,\cdot,\cdot,\cdot): \Sigma^* \times N \times \Sigma^* \times \Sigma^* \times \Sigma^* \to$$
$$\to \Sigma^* \times N \times N \times \Sigma^* \times \Sigma^*$$
$$(\alpha^*)^{-1}(c,b,w,f,t) = (\sigma(c,b,b+|\rho(f,t,w)|,w), b, b+|w|, f, t)$$

5. A Tool Which Implements the Operators of the Model

To experiment the model introduced in the previous Sections we developed a set of functions that could be added to a well- known text editor:

Xemacs. These functions allow the programmers to perform transformations on their code in the way the operators describe them. The transformations are recorded in a file and other functions interpret this file enabling a navigation through all the versions of the code highlighting the kind of transformation that led from one version to another .

The operator to apply is selected from a menu and the position parameters are specified using the mouse.

The file containing the history of a program records the date and time of change, the operator applied and its parameters.

For the operators that insert new characters in the code (Insert and Substitute) it is possible to choose if these characters have to be taken from the keyboard or from another file. When the string is taken from a file, in the history file it is the name of the file together with the positions that bound the inserted characters which are recorded, not the characters. In this situation it is very important to record the exact time of change, since it is necessary to accede to the right version of the file which the characters were taken from to rebuild correctly the code when navigating through the changes. If the source file has changed since the extraction of the inserted string, then the navigation functions are going to be recursively applied to this file for getting its version at the time of the extraction.

A possible way for the implementation of the transformations is the simple application of the definitions given for the operators plus some auxiliary functions for the management of windows and files. This approach is only theoretically possible. In fact, the recursive nature of some of the functions used to treat the strings made them very expensive in terms of used memory. For example the function used to calculate the length of a string is used directly or indirectly in almost every other operation. For example, if we recall its definition ($|\varepsilon| = 0$, $|s| = 1 + |\mathrm{LTRUNC}(s)| \quad \forall s \neq \varepsilon$) we can see that the function has to be applied 1001 times to calculate the length of a string of 1,000 characters. For this reason the operators will use the operations on strings pre-defined in the Elisp language used to extend the functionalities of Xemacs, and not the functions defined in Section 2.

6. Conclusion

In this work we introduced a model that formally describes the evolution of a program on which some changes are performed in order to improve, maintain or re-use it.

In this model the code is represented as a string without any syntactic or semantic value. This choice was made in order to get the most generality for the model which was completely independent of the used programming language.

The first step has been the definition of a set of functions which allows to manipulate the strings concatenating them, extracting some parts, or calculating their length. Then five operators based on these functions were introduced.

The operators formally describe the transformations more frequently applied to a program:

- insertion of new parts;
- deletion of part of the code;
- substitution of a chunk of the code for another one;
- replacement of all the occurrences of a given string by another one;
- application of some function taking as its argument part of the original code.

The multiplicity of the transformations usually encountered when modifying a program drove to look for some properties which allowed to reduce the number of operations necessary for getting from one version of the code to a subsequent one.

Then the introduced operators have been extended in order to render them invertible. This allows to rebuild older versions of a program starting from the newer ones. Some symmetry has been found on inverting the operators: the insertion and the deletion are one the inverse of the other, while the substitution coincides with its inverse.

Finally a set of Lisp functions have been written to add to a text editor the capability of performing the transformations described by the model and of navigating through the different versions of the code.

This model can be useful for several reasons.

It can help in the understanding of a program, because realizing the changes undergone by the code it makes it easier to identify the functionalities associated with different parts of a program.

Keeping trace of the old changes can help in trying to foresee the effects of new changes, or to identify the variant parts of a program in a domain analysis context.

It can be useful in testing and documentation phases because it can reduce the bulk of work to do, also allowing the reuse of previous results.

The introduced properties make it easier to verify the properties of the code since a reduction in the number of changes can simplify the understanding of the goal of transformations.

It would be interesting to try to apply this model to a real case in order to evaluate its effective potential. In the future the model will be used to describe a set of transformations for some programming language, which do not alter the behaviour of a program. Then we will try to identify some properties that can be found in a program and the way the operators applied on the code affect them.

Appendix

A sample demonstration

We want to demonstrate that:

if $n \le m \le n+s$ then $\iota(\iota(c,n,s),m,t) = \iota(c,n,\iota(s,m-n+1,t))$.

This is done to show that if the hypothesis is true then the right and the left sides of the equation are equal.

From the definition of ι we have for the left side:

$$\iota(\iota(c,n,s),m,t) = \mathrm{LSTRING}(\iota(c,n,s),m-1) \cdot t \cdot \mathrm{RSTRING}(\iota(c,n,s),|\iota(c,n,s)|-m+1)$$

If we consider separately the first and the last member of the concatenation we have:

$$\mathrm{LSTRING}(\iota(c,n,s),m-1) = \mathrm{LSTRING}(\mathrm{LSTRING}(c,n-1)\cdot s\cdot \mathrm{RSTRING}(c,|c|-n+1),m-1)$$

This means that we have to take the first m-1 characters starting from the left of a string obtained by concatenating three strings. For this hypothesis it is $n \le m$ and this implies that $n-1 \le m-1$. This means that all the characters of the first substring belong to the result of LSTRING.

For the hypothesis it is $m \le n+|s|$ and then $m-n \le |s|$. Then all the characters that are extracted from the other two substrings, which are m-1-(n-1)=m-n all belong to s.

This means that:

$$\mathrm{LSTRING}(\iota(c,n,s),m-1) = \mathrm{LSTRING}(c,n-1)\cdot \mathrm{LSTRING}(s,m-n)$$

In the same way for the third string:

$$\mathrm{RSTRING}(\iota(c,n,s),|\iota(c,n,s)|-m+1) = \mathrm{RSTRING}(\mathrm{LSTRING}(c,n-1)\cdot s\cdot \mathrm{RSTRING}(c,|c|-n+1),|c|+|s|-m+1)$$

$|c|+|s|-m+1$ characters have to be taken from the right side of the concatenation of three substrings. If we calculate the difference between the length of the extracted substring and the length of the third substring we obtain $|c|+|s|-m+1-(|c|-n+1) = |s|-n+m \ge 0$ for the starting hypothesis $m \le n+|s|$.

This implies that the whole third substring is contained in the result of RSTRING and that $|s|$-n+m more characters have to be taken from the other two substrings. The hypothesis $n \le m$ implies that $|s|-n+m \le |s|$, then all the remaining characters are extracted from s:

$$\mathrm{RSTRING}(\iota(c,n,s),|\iota(c,n,s)|-m+1)=$$
$$\mathrm{RSTRING}(s,|s|+n-m)\cdot$$
$$\mathrm{RSTRING}(c,|c|-n+1),|c|+|s|-m+1)$$

We can finally write:

$$\iota(\iota(c,n,s),m,t)=$$

$$\mathrm{LSTRING}(c,n-1)\cdot\mathrm{LSTRING}(s,m-n)\cdot t\cdot$$
$$\mathrm{RSTRING}(s,|s|+n-m)\cdot$$
$$\mathrm{RSTRING}(c,|c|-n+1),|c|+|s|-m+1)$$

If we consider the right side of the equation deduced from the definition of Insert we obtain the same result:

$$\iota(c,n,\iota(s,m-n+1,t))=$$

$$\iota(c,n,\mathrm{LSTRING}(s,m-n)\cdot t\cdot$$
$$\mathrm{RSTRING}(s,n+|s|-m))=$$

$$\mathrm{LSTRING}(c,n-1)\cdot\mathrm{LSTRING}(s,m-n)\cdot t\cdot$$
$$\mathrm{RSTRING}(s,|s|+n-m)\cdot$$
$$\mathrm{RSTRING}(c,|c|-n+1),|c|+|s|-m+1)$$

REFERENCES

1. BOYLE, J.M. and MURALIDHARAN, M.N., **Program Reusability Through Program Transformation,** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-10, No. 5, September 1984, pp. 574-588.
2. LOVEMAN, D.B., **Program Improvement by Source-to-Source Transformation,** JOURNAL OF THE ACM, Vol. 24, No. 1, January 1977, pp. 121-145.
3. ARSAC, J.J., **Syntactic Source to Source Transforms and Program Manipulation,** COMMUNICATIONS OF THE ACM, Vol. 22, No.1, January 1979, pp. 43-54.
4. BURSTALL, R.M. and DARLINGTON, J., **A Transformation System for Developing Recursive Programs,** JOURNAL OF THE ACM, Vol. 24, No. 1, January 1977, pp. 44-67.
5. PARTSCH, H., **An Exercise in the Transformational Derivation of An Efficient Program by Joint Development of Control and Data Structure,** Science of Computer Programming, Vol. 1, 1984, pp. 1-35.
6. BUSH, E., **The Automatic Restructuring of COBOL,** Proceedings of the Conference on Software Maintenance 1985, Los Alamitos, CA, USA, IEEE Computer Society, 1985, pp. 35-41.
7. PARTSCH, H. and STEINBRUGGEN,R., **Program Transformation Systems,** COMPUTER SURVEYS, Vol. 15, September 1993, pp. 199-236.
8. WEISER, M., **Program Slicing,** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-10, No. 7, July 1984, pp. 352-357.
9. FREAK, R.A., **A Fortran to Pascal Translator,** SOFTWARE PRACTICE AND EXPERIENCE, Vol. 11, July 1981, pp. 717-732.
10. IRLIK, J., **Translating Some Recursive Procedures into Iterative Schemes,** in B.Robinet (Ed.) Programmation, Paris, DUNOD, 1977, pp. 39-52.
11. WEGBREIT, B., **Goal-Directed Program Transformation,** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-2, No. 2, June 1976, pp. 69-80.
12. ALLEEN, F.E. and COCKE, J., **A Catalogue of Optimizing Transformations,** in R. Rustin (Ed.) Design and Optimization of Compilers, Englewood Cliffs, NJ, USA, PRENTICE HALL, 1972, pp. 1-30.
13. MERKS, E.A.T., DYCK, J.M. and CAMERON, R.D., **Language Design for Program Manipulation,** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-18, No. 1, January 1992, pp. 19-32.
14. BACKUS, J., **Can Programming Be Liberated from Von Neumann Style? A**

Functional Style and Its Algebra of Programs, COMMUNICATIONS OF THE ACM, Vol. 21, No. 8, August 1978, pp. 613-641.

15. OVERSTREET,C.M., CHEN, J. and BYRUM, F., **Program Maintenance by Safe Transformations,** Proceedings of the Conference of Software Maintenance - 1988, Washington, DC, USA, IEEE Computer Society Press, 1988, pp. 118-123.

16. AIKEN, A., WILLIAMS, J.H. and WIMMERS, E.L., **Program Transformations in the Presence of Errors,** Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages, New York, NY, USA, ACM, 1990, pp. 210-217.

17. RATCLIFF, B., **Re-affirming Support for A Transformational Approach to Software Development: Theory and Practice,** Proceedings of the 15th Annual International Computer Software and Application Conference, Los Alamitos, CA, USA, IEEE Computer Society Press, 1991, pp. 493-498.

18. DEVIENNE, P. and LEBEGUE, P., **Prolog Program Transformations and Meta-Interpreters,** Logic Program Synthesis and Transformation, Proceedings of LOPSTR 91, Berlin, Germany, SPRINGER-VERLAG, 1992, pp. 238-251.

19. AMEUR,Y.A., **Program Transformations Directed by the Evaluation of Non Functional Properties,** Logic Program Synthesis and Transformation, Proceedings of LOPSTR 91, Berlin, Germany, SPRINGER-VERLAG, 1992, pp. 267-299.

20. CHEATHAM, Jr, T.E., **Reusability Through Program Transformation,** IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 589-594.

21. PETTOROSSI, A. and PROIETTI, M., **Rules and Strategies for Program Transformation,** in B.Möller, H.Partsch and S. Schuman (Eds.) Formal Program Development. Lecture Notes in Computer Science, Berlin, Germany, SPRINGER-VERLAG, 1993, pp. 263-304.

22. BOYLE, J.M. and HARMER,T.J., **Practical Transformations of Functional Programs for Efficient Execution: A Case Study,** in B. Möller, H. Partsch and S. Schuman (Eds.) Formal Program Development. Lecture Notes in Computer Science, Berlin, Germany, SPRINGER-VERLAG, 1993, pp. 62-88.

23. BOYLE, J.M., **Abstract Programming and Program Transformation-An Approach To Reusing Program,** in T.J. Biggerstaff and A.J.Perlis (Eds.) Software Reusability, Vol. 1, New York, USA, ADDISON-WESLEY PUBLISHING COMPANY, 1989, pp. 361-413.

24. BOYLE, J.M., **Program Adaptation and Program Transformation,** in R. Ebert, J. Lueger and L. Goecke (Eds.) Practice in Software Adaptation and Maintenance, Amsterdam, The Netherlands, NORTH HOLLAND PUBLISHING CO., 1980, pp. 3-20.

25. WIRTH, N., **Program Development by Stepwise Refinement,** COMMUNICATIONS OF THE ACM, Vol. 14, April 1971, pp. 221-227.

26. BOYLE, J.M., **Lisp to Fortran-Program Transformation Applied,** in P.Pepper (Ed.) Proc. Workshop on Program Transformation and Programming Environments, Nato ASI Series F, COMPUTER AND SYSTEMS SCIENCES 8, Berlin, Germany, Springer-Verlag, 1984, pp. 291-298.

27. KELLNER, M.J., **Software Process Modelling At SEI,** in Proceedings of the 1988 Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society Press, 1988, p. 78.

28. DONG, J. WILD, C. and MALY, K., **A Software Development and Evolution Model Based on Decision-Making,** the 3rd International Conference on Software Engineering and Knowledge Engineering, Stokie, IL, USA, Knowledge System Inst., 1991, pp. 9-14.

29. GERHART, S., **Formal Methods: An International Perspective,** the 13th International Conference on Software

Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press, 1991, pp. 36-37.

30. HANLON, J.G. and INGLIS, I.R., **Fitness for Purpose Requires the Anticipation of Change (Software Development),** IEE Colloquium on Designing Quality into Software Based Systems, London, UK, IEE, 1991, pp. 3/1-3.

31. MADHAVJI, N.H., **The Prism Model of Change,** the 13th International Conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press, 1991, pp. 166-177.

32. LIN, J.M., **Issues On Deterministic Transformation of Logic-Based Program Specification,** Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence, Los Alamitos, CA, USA, IEEE Computer Society Press, 1990, pp. 603-609.

33. YAMADA, H. and TEZUKA, Y., **Functional Model Based Assistance for Modifying Software,** System Computer Jpn (USA), Vol. 22, No. 13, 1991, pp. 82-92.

34. ROTENSTREICH, S.,**Transformational Approach to Software Design,** Inf. Software Technology (UK), Vol. 34, No. 2, February 1992, pp. 106-116.

35. GOGUEN,J.A., **An Algebraic Approach to Refinement,** VDM '90. VDM and Z - Formal Methods in Software Development. 3rd International Symposium of VDM Europe Proceedings, Berlin, Germany, SPRINGER-VERLAG, 1990, pp. 12-28.

36. ROTENSTREICH, S., **Enhancement Through Design Transformations: A Retroactive Case Study,** J. SOFTWARE MAINTENANCE RESEARCH PRACTICE, Vol. 2, No. 4, December 1990, pp. 193-208.

37. GOLDENBERG, A., **Reusing Software Developments,** SIGSOFT Software Engineering Notes, Vol. 15, No. 6, December 1990, pp. 107-119.

38. VARDHARAJAN, V., **A Formal Approach To System Design and Refinement,** COMPEURO '90 Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press, 1991, pp. 544-545.

39. RATTRAY, C., **An Evolutionary Software Model,** Mathematical Structures for Software Engineering, based on the Proceedings of a Conference, Oxford, UK, Clarendon Press, 1991, pp. 185-192.

40. PURTILO,J.M. and ATLEE, J.M., **Module Reuse by Interface Adaptation,** SOFTWARE PRACT. EXP., Vol. 21, No. 6, June 1991, pp. 539-556.

41. BURSON, S. KOTIK, G.B. and MARKOSIAN, L.Z., **A Program Transforma-tion Approach to Automating Software Re-engineering,** Proceedings of the 14th Annual International Computer Software and Applications Conference, Los Alamitos, CA, USA, IEEE Computer Society Press, 1991, pp. 314-322.

42. GLASSON, B.C., **Model of System Evolution,** INF. SOFTWARE TECHNOLOGY, Vol. 31, No. 7, September 1989, pp. 351-356.

43. KELLER, S.E., **Grammar-Based Program Transformation,** Proceedings of the Conference on Software Maintenance 1988, Washington, DC, USA, IEEE Computer Society Press, 1988, pp. 110-117.

44. SCACCHI, W., **Modeling Software Evolution: A Knowledge Based Approach,** 'Representing and Enacting the Software Process' Proceedings of the 4th International Software Process Workshop, New York, NY, USA, ACM, 1989, pp. 153-155.

45. LEHMAN, M.M., **Modes of Evolution (Software Engineering),** 'Iteration in the Software Process'. Proceedings of the 3rd International Software Process Workshop, Washington, DC, USA, IEEE Computer Society Press, 1987, pp. 29-32.

46. FICKAS, S.F., **Automating the Transformational Development of Software,** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 11, No. 11, November 1995, pp. 1268-1277.

47. ARANGO,G. BAXTER, I., FREEMAN, P. and PIDGEON, C., **TTM: Software Maintenance By Transformation,** IEEE SOFTWARE, Vol. 3, No. 3, May 1986, pp. 27-39.

48. DAVIS, M.D. and WEYUKER, E.J., Computability, Complexity and Languages - Fundamentals of Theoretical Computer Science, San Diego, CA, USA, ACADEMIC PRESS INC., 1983.