Framework for a Formal Representation of Code Evolution

Giancarloucci, Marco Ronchetti, Carl Uhrlik
Francesco Baruchelli, Guido Cardino, Andrea Valerio
LII/DISA, Università di Trento
Via Zeni 8, I-38068 Rovereto, Italia
E-mail: {FirstName.LastName}@lii.unitn.it
Phone: +39-464-443140 Fax: +39-464-443141

Abstract
This paper introduces a model that formalizes the later stages of the software life cycle, in which the code is modified. The reasons that led to this model are analyzed, then the model itself is presented. The model consists of a series of operators that act on the code considered as a simple text file, without any particular semantic meaning or structure. First some functions on strings and integers on which the operators are based, then the operators are introduced. Finally the operators are extended to allow their inversion. A tool has been built that implements the characteristics of the model, allowing the user to keep track of the changes occurred to his code.

Keywords: program transformation, software model, software engineering.

1 Introduction

Software does not deal with static entities, but is something that is continuously changing and evolving. The changes may be done for the maintenance of the software, for an improvement of some of its characteristics, or even for developing a completely new program from an existing one.

There are many models and formalisms, some of them very complex, that aim to give a rigorous description of the software evolution process.

Most of these models consider the process leading from the user’s expectations to the final code, going through some intermediate steps, such as the product requirements definition and the system design.

This paper introduces a simple model that describes only the later stages of the software life cycle after the initial implementation of a piece of software: it describes the history of the considered code, intended as the series of transformations that the source code undergoes for its maintenance or its evolution, or for its reuse in a new product. Then a tool based on the model that keeps track of these changes is described.

It is important to keep track of these transformations, with the reasons that made them necessary, for a better understanding of the program being considered. This importance seems evident if we consider the case of more programmers working on the same code, or a programmer that has to look at some piece of code he has not been working on for some time.

A knowledge of this history can help in understanding how a program works because it can be used to make inductive inferences: if we have the history $X_1, X_2, X_3, ..., X_n$ (where each $X_i$ is a version of the code) and as well the effects that go with the transformations $X_1 \rightarrow X_2, X_2 \rightarrow X_3, ..., X_{n-1} \rightarrow X_n$, these can be useful in trying to predict the effect of a further transformation $X_n \rightarrow X_{n+1}$. On the other side, the knowledge of the software’s history can help to identify the parts of the code more subjected to changes. This characteristic can be very useful, for instance, in a domain analysis method to predict and characterize the domain variability.
Knowing the history of a program can be useful in testing, too. For instance a piece of code could have been inserted from another program that was known to behave correctly, so it is supposed to work properly in this implementation, too, and the tester can focus his attention only on completely new lines of code, or on the interaction between new lines and the inserted chunk.

Furthermore formalizing the process of program editing can help the editing process itself supplying the programmer with a paradigm according to which to classify his/her action.

2 Definition of the model

The model introduced in this paper is very simple. It does not have the aim to give a very rigorous description of the changes in software, but only to allow the construction of a tool that can help in keeping track of the history of a piece of code. It simply describes the transformations of the code in a formal way, leaving to the programmers all the questions regarding the correctness of the software after the changes are made.

The model consists of five simple operators that act on the code represented as a plain text file, then as a simple string, without any particular semantic meaning or structure. These operators describe the typical operations that are made when altering a chunk of code. They are:

- Insert (ι)
- Delete (χ)
- Substitute (σ)
- Replace (ρ)
- Apply (α)

All these operators act on strings representing the source code being modified. The way they act is specified through a series of parameters that are passed to the operators themselves.

Acting on strings, they are based on a simple set of operations on strings and integers that will be introduced in section 2.2.

2.1 Description of the operators

- Insert (ι)

   This operator allows to add new lines (or simply new characters) to the code in any position. The parameters must specify the characters to be inserted, the string (then the program) where to insert them and their position in the string.

- Delete (χ)

   This operator allows to cut a chunk of code, and the parameters needed are obviously the string the chunk to be deleted belongs to and the characters' first and last position in the string.

- Substitute (σ)

   This operator allows to substitute a chunk of code with another one. It can be seen as an application of χ followed by an application of ι. The parameters needed are: original string, starting and ending position of the substring to substitute, new substring to insert.
• Replace ($\rho$)

This operator allows a kind of substitution different from the one described for the previous operator. It replaces all the occurrences of a given substring or a given regular expression with a new substring. The parameters needed are: original string, substring to substitute, substring to insert.

• Apply ($\alpha$)

This operator allows to apply a particular function of the language used in the file being modified to a chunk of the code considered. For example, if we want to repeat $n$ times a series of statements, this can be done by putting the lines in a for loop through this operator.

The whole history of a program can be described using an opportune combination of the operators introduced above, even the creation of a completely new program from scratch can be modelled in this way through the use of the operator $\iota$.

2.2 Operations on strings

Let us define as $\Sigma$ the set of ASCII characters of which our files are made up of, and $\Sigma^*$ the set of all the possible combinations of the characters in $\Sigma$, then the set of all the possible strings, including the void one represented by $\varepsilon$. Then let $\Sigma_\perp$ be the union of $\Sigma$ and the void string $\varepsilon$. $\mathbb{N}$ is the set of all the integers starting from 0. The following are the operations allowed on strings on which the operators are based.

• Concatenate: this function returns the string obtained concatenating two or more strings.

$$CONCAT(\cdot) : (\Sigma^*)^* \to \Sigma^*$$

$$CONCAT(\varepsilon) = \varepsilon$$

$$CONCAT(u_1, \ldots, u_{m+1}) = CONCAT(u_1, \ldots, u_m)u_{m+1}$$

• Left End: returns the first character at the left end of the string.

$$LTEND(\cdot) : \Sigma^* \to \Sigma_\perp$$

$$LTEND(\varepsilon) = \varepsilon$$

$$LTEND(s) = \text{the first char starting from the left in s (s[1] if we consider a string as a vector of characters with index starting from 1).}$$

• Left Truncate: returns the string without its Left End.

$$LTRUNC(\cdot) : \Sigma^* \to \Sigma^*$$

$$LTRUNC(\varepsilon) = \varepsilon$$

$$LTRUNC(s) = w \text{ in } \Sigma^* \text{ such that } CONCAT(LTEND(s), w) = s$$

• Length: gives the number of characters in a string.

$$|\cdot| : \Sigma^* \to \mathbb{N}$$

$$|\varepsilon| = 0$$

$$|s| = 1 + |LTRUNC(s)|$$

• Middle Character: returns the character in a defined position in a string.

$$MID(\cdot, \cdot) : \Sigma^* \times \mathbb{N} \to \Sigma_\perp$$

$$MID(\varepsilon, n) = \varepsilon$$

$$MID(s, 1) = LTEND(s)$$
\[ \text{MID}(s, 0) = \epsilon \]
\[ \text{MID}(s, n) = \epsilon \forall n > |s| \]
\[ \text{MID}(s, n) = \text{MID}(\text{LTRUNC}(s), n - 1) \]

- Right End: like Left End, but starting from the right.
\[ \text{RTEND}(\cdot) : \Sigma^* \to \Sigma_\perp \]
\[ \text{RTEND}(\epsilon) = \epsilon \]
\[ \text{RTEND}(s) = \text{MID}(s, |s|) \]

- Right Truncate: like Left Truncate, but keeps the left side of the string.
\[ \text{RTRUNC}(\cdot) : \Sigma^* \to \Sigma^* \]
\[ \text{RTRUNC}(\epsilon) = \epsilon \]
\[ \text{RTRUNC}(s) = w \text{ in } \Sigma^* \text{ such that } \text{CONCAT}(w, \text{RTEND}(s)) = s \]

- Substring: returns the substring of a string starting from one position (the character at that position is included in the substring) and ending with another position (the character at that position is excluded).
\[ \text{SUBSTRING}(. , . , .) : \Sigma^* x N x N \to \Sigma^* \]

(the first integer is the starting position, the second the ending position)
\[ \text{SUBSTRING}(\epsilon, n, m) = \epsilon \]
\[ \text{SUBSTRING}(s, n, m) = \epsilon \forall n, m \text{ in } N \text{ such that } m \leq n \]
\[ \text{SUBSTRING}(s, 0, m) = \text{SUBSTRING}(s, 1, m) \]
\[ \text{SUBSTRING}(s, n, m) = \text{SUBSTRING}(s, n, |s| + 1) \forall m \text{ in } N \text{ such that } m > |s| + 1 \]
\[ \text{SUBSTRING}(s, n, n + 1) = \text{MID}(s, n) \]
\[ \text{SUBSTRING}(s, n, m) = \text{CONCAT} \left( \text{SUBSTRING}(s, n, n + 1), \text{SUBSTRING}(s, n + 1, m) \right) \]

- Left Substring: returns the first n characters of a string starting from the left.
\[ \text{LSTRING}(., .) : \Sigma^* x N \to \Sigma^* \]
\[ \text{LSTRING}(s, n) = \text{SUBSTRING}(s, 0, n + 1) \]

- Right Substring: like Left Substring but starting from the right.
\[ \text{RSTRING}(., .) : \Sigma^* x N \to \Sigma^* \]
\[ \text{RSTRING}(s, n) = \text{SUBSTRING}(s, |s| - n + 1, |s| + 1) \]

- Search Substring: returns the position of the first occurrence of a given substring in a string starting from a specified position (it returns 0 if the substring is not present). It makes use of an auxiliary function that searches the substring starting from the first character and has a different behaviour when the substring is not found (returns |s| + 1, where s is the string in which to search, and not 0).
\[ \text{SAUX}(., .) : \Sigma^* x \Sigma^* \to N \]

The substring to look for is the second parameter.
\[ \text{SAUX}(\epsilon, t) = 1 \]
\[ \text{SAUX}(s, t) = 1 \text{ if } \text{LSTRING}(s) = t \]
\[ \text{SAUX}(s, t) = 1 + \text{SAUX}(\text{LTRUNC}(s), t) \]

The function that will be used in the representation is defined as follows:
\[ \text{SEARCH}(\ldots) : \Sigma^* x \Sigma^* x N \rightarrow N \]
\[ \text{SEARCH}(\varepsilon; t, n) = 0 \]
\[ \text{SEARCH}(s; t, 0) = \text{SEARCH}(s; t, 1) \]
\[ \text{SEARCH}(s; t, n) = n \text{ if } \text{SUBSTRING}(s; n, n + |t|) = t \]
\[ \text{SEARCH}(s; t, n) = 0 \text{ if } [n + \text{SAUX}(\text{RSTRING}(s; |s| - n), t)] > |s| \]
\[ \text{SEARCH}(s; t, n) = n + \text{SAUX}(\text{RSTRING}(s; |s| - n), t) \text{ else} \]

With these operations on strings and regular expressions, all the operators can be defined.

2.3 Definition of the operators

In this section, we formalize the operators described in section 2.1 through the definition of their domain and range and the way they act as operations on strings.

- Insert (\(\iota\))

\[ \iota(\ldots) : \Sigma^* x N x \Sigma^* \rightarrow \Sigma^* \]

The first parameter is the original string; the second, the insertion point; and the third, the string to insert.

\[ \iota(c; 0, s) = \iota(c; 1, s) \]
\[ \iota(c; n, s) = \iota(c; |c| + 1, s) \text{ for each } n \in \mathbb{N} \text{ such that } n > |c| + 1 \]
\[ \iota(c; n, s) = \text{CONCAT}(\text{LSTRING}(c; n - 1), s, \text{RSTRING}(c; |c| - n + 1)) \text{ else} \]

Example: \(\iota(\text{"It is raining"}, 3, \text{"seems that it"}) = \text{"It seems that it is raining"}\)

- Delete (\(\chi\))

\[ \chi(\ldots) : \Sigma^* x N x N \rightarrow \Sigma^* \]

The first parameter is the original string; the second, the beginning position of the chunk to delete; and the third, the ending position of the chunk. (Here, and in the rest of the paper, the character at the beginning position is included in the chunk to be deleted, while the character at the ending position will not be deleted).

\[ \chi(c; b, e) = \text{CONCAT}(\text{LSTRING}(c; b - 1), \text{RSTRING}(c; |c| - e + 1)) \]

Example: \(\chi(\text{"It seems that it is raining"}, 3, 17) = \text{"It is raining"}\)

- Substitute (\(\sigma\))

\[ \sigma(\ldots) : \Sigma^* x N e m x N x \Sigma^* \rightarrow \Sigma^* \]

The first parameter is the original string; the second, the beginning position of the chunk to substitute; the third, the ending position of the chunk; and the fourth, the string to insert.

\[ \sigma(c; b, e, s) = \text{CONCAT}(\text{LSTRING}(c; b - 1), s, \text{RSTRING}(c; |c| - e + 1)) \]

Example: \(\sigma(\text{"I think that it is not raining"}, 3, 8, \text{"hope"}) = \text{"I hope that it is not raining"}\)

- Replace (\(\rho\))

\[ \rho(\ldots) : \Sigma^* x \Sigma^* x \Sigma^* \rightarrow \Sigma^* \]

The first parameter is the original string; the second, the string to substitute; and the third, the string to insert.

\[ \rho(s; \varepsilon, m) = s \]
\[ \rho(s; t, m) = s \text{ if } \text{SEARCH}(s; t, 0) = 0 \]
\[ \rho(s, t, m) = \text{CONCAT}(LSTRING(s, SEARCH(s, t, 0) - 1), m, \rho(RSTRING(s, |s| - \text{SEARCH}(s, t, 0) + 1 - |t|), t, m)) \text{ else} \]

Example: \( \rho("The more I sleep the more I'd like to sleep", "sleep", "work") = "The more I work the more I'd like to work" \)

- **Apply \((\alpha)\)**

  \[ \alpha(\ldots, \ldots, \ldots, NxN, x) : \Sigma^* \times N \times N \times \Sigma^* \times x \rightarrow \Sigma^* \]

  The first parameter is the original string; the second, the beginning position of the chunk to which the function will be applied; the third, the ending position of the chunk; the fourth, the function to apply; and the fifth, the argument of the function.

  \[ \alpha(c, b, e, f, t) = \sigma(c, b, e, \rho(f, t, SUBSTRING(c, b, e))) \]

  Example:

  \[ \alpha("int i; \\n i++; \\n nexit(-2); \\n \"n", 8, 11, "if(x > 10)\n x; \\n nelse\n x; \\n exit(-1)\"", "\") = "int i; \\n i++; \\n nelse\n i++; \\n exit(-1); \\n exit(-2);" \]

  (because here \n is for new line)=

  \[ \begin{align*}
  &\text{int i;} \\
  &\text{if (i ++ > 10)} \\
  &\quad i ++ ; \\
  &\text{else} \\
  &\quad \text{exit(-1);} \\
  &\text{exit(-2);} \\
  \end{align*} \]

2.4 **Inverse operators**

The operators defined in section 2.3 are not invertible, so it is impossible to return to an earlier version of the modified file using only the result of the operator's application.

This can be achieved with an expansion of the range of the operators to give all the necessary parameters to go back after a change. The core of these new operators are the same previously defined with the addition of small parts to grant invertibility.

Here the star indicates the extended version of the operator, while the simple version is the one defined in section 2.3.

- **Insert**

  \[ \iota^* (\ldots, \ldots, N) : \Sigma^* \times N \times \Sigma^* \rightarrow \Sigma^* \times N \times N \]

  (Original string, Insertion point, String to insert) \(\rightarrow\) (Modified string, Beginning position of the inserted string, Ending position of the inserted string)

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

  The inverse of this operator can be defined in the following way:

  \[ (\iota^*)^{-1}(\ldots, \ldots, \ldots) : \Sigma^* \times N \times N \rightarrow \Sigma^* \times N \times \Sigma^* \]

  \[ (\iota^*)^{-1}(c, b, e) = (\chi(c, b, e), b, SUBSTRING(c, b, e)) \]

- **Delete**

  \[ \chi^* (\ldots, \ldots, N) : \Sigma^* \times N \times N \rightarrow \Sigma^* \times N \times \Sigma^* \]

  (Original string, Beginning position of the string to remove, Ending position of the string to delete) \(\rightarrow\) (Modified source, Starting position of the deleted string, Deleted string)

  \[ \chi^*(c, b, e) = (\chi(c, b, e), b, SUBSTRING(c, b, e)) \]
The inverse is:

\((\chi^*)^{-1}(c, n, s) = (\iota(c, n, s), n, n + |s|)\)

It can be noted that the domain of \(\iota^*\) is the same set of the range of \(\chi^*\) and that the range of \(\iota^*\) is the same set of the domain of \(\chi^*\). Then, looking at the definitions of the operators and their inverse, it can be stated that:

\((\iota^*)^{-1}(c, b, e) = \chi^*(c, b, e)\)

and

\((\chi^*)^{-1}(c, n, s) = \iota^*(c, n, s)\)

As one could think, the operations of inserting a new string in a text or deleting a substring from it are the same in the formalization proposed and not only in an intuitive way.

- Substitute

  \(\sigma^*(c, b, e, s) = (\sigma(c, b, e, s), b + |s|, SUBSTRING(c, b, e))\)

The inverse is:

\((\sigma^*)^{-1}(c, i, f, w) = (\sigma(c, i, f, w), i, i + |w|, SUBSTRING(c, i, f))\)

In this case the operator has the same set as its domain and as its range, and from the two definition it can be seen how the operator is equal to its inverse:

\((\sigma^*)^{-1}(c, i, f, w) = \sigma^*(c, i, f, w)\)

3 Description of the tool

To validate the model and to test its effective utility, a tool based on it has been developed.

The model introduces five operators to keep track of the history of a program; the purpose of the tool is to record in some way the sequence of the changes made to a file (i.e., the operators applied with their parameters), and then to allow the programmer to see all the versions of the file being edited with the operation that led to them.

The tool consists of a simple text editor with the addition of the functions performed by our model's operators, and the capability to trace the history of a file through a series of steps forward or backward starting from the first or from the last version of the file itself.

The way chosen to implement this editor is to add these capabilities to Xemacs (this editor was chosen because it is very easy to customize).

Each time an operator is applied, an associated function has to modify the text being edited, but must record the change made in some way, too. This is done through the use of two additional files: in the file \(\text{<file}_\text{name>}.\text{first}\) is stored the first version of the code to which the operators were applied, while in the file \(\text{<file}_\text{name>}.\text{history}\) there is a record for each operator applied and its parameters. Reading these files, the tracing functions of the editor can skip backwards and forwards through the sequence of the versions of the file simply reapplying the operators recorded.

A change in the file is shown by these functions comparing in two different windows (or three if a third file is involved as source for some parameter) two successive versions of the code, in which the changed chunks are indicated with different colours (for example the deleted chunk is shown in red in the "before" version, while the newly inserted chunk is shown in green in the
"after" version, where "before" and "after" refer to the versions before and after the change was made).

A parameter not considered in the operators, but very important in their recording, is the time (the date and the hour) at which the operator was applied. In fact, if a parameter depends upon another file, it is essential to know the exact version of the file from which that parameter has been taken in order to be able to trace the change correctly, because that file could have been changed since that time, too. Comparing the time of the change with the time the file was last changed, the tracing function can decide if the parameter can be drawn from the current version of the file, or if an earlier version has to be rebuilt through the use of the operators and the history files again. Of course this procedure can be repeated recursively for rebuilding a version of the parameter’s source file if this has in its history some operator that refers to another file again.

4 Conclusion and future work

The model seems to fit its goals. It describes correctly the changes a generic piece of software can undergo for its maintenance, improvement or reuse.

Developing the tool, some additional information had to be added to the parameters of some operators to allow backward tracing. This was done for the operators that are not invertible, yet: Replace and Apply.

Future work will look to a further development of the model and the tool. A thing to be eliminated is the “noise” generated when making a change in a file without using the operators (for example when correcting a syntax error using the Backspace key). These kinds of changes modify the file without being recorded, so the subsequent operators act on a version of the code different from the one recorded.

Some improvements in the model will be the definition of the missing inverse operators and some check of the correctness, at least syntactic, of the code after the change has been made.

Another problem is given by the size of the history files, that have to be compressed some way.

Finally, the tool will be integrated, in the view of reuse, with a software library system that will allow to look for some piece of code to use in the software being developed. The information accessible through this system will be distributed, so a new problem arises: the clock processes on two different machines may not be synchronized for various reasons, and this could put at risk the validity of the information of the history files.