

# Modeling Clones Evolution Through Time Series

G. Antoniol\*, G. Casazza\*\*, M. Di Penta\*, E. Merlo \*\*\*

antoniol@ieee.org, gec@unisannio.it, dipenta@unisannio.it, ettore.merlo@polymtl.ca

(\*) University of Sannio, Faculty of Engineering - Benevento, Italy

(\*\*) University of Naples "Federico II", DIS - Naples, Italy

(\*\*\*) École Polytechnique de Montréal - Montréal, Canada

## Abstract

*The actual effort to evolve and maintain a software system is likely to vary depending on the amount of clones (i.e., duplicated or slightly different code fragments) present in the system.*

*This paper presents a method for monitoring and predicting clones evolution across subsequent versions of a software system. Clones are firstly identified using a metric-based approach, then they are modeled in terms of time series identifying a predictive models.*

*The proposed method has been validated with an experimental activity performed on 27 subsequent versions of mSQL, a medium-size software system written in C. The time span period of the analyzed mSQL releases covers four years, from May 1995 (mSQL 1.0.6) to May 1999 (mSQL 2.0.10). For any given software release, the identified models was able to predict the clone percentage of the subsequent release with an average error below 4%. An higher prediction error was observed only in correspondence of major system redesign.*

**Keywords:** software evolution, clone analysis, time series

## 1. Introduction

It is widely recognized that software systems must evolve to meet user ever changing needs [16, 17]. Several evolution driving factors may be identified: new functionalities added, lack of software quality, lack of overall system performance, software portability (on new software and hardware configurations, i.e., new platforms) and market opportunities are just few examples.

When evolving a software system, developers may decide to copy an entire working sub-system, then rename all the functions and start modifying the software. This technique ensures they will not have any unplanned effect on

the original piece of code they have just copied. However, this evolving practice promotes the appearing of clones.

As a part of a larger study we are investigating the influence of software evolution on software size with special regards to duplicated and/or slightly modified code also known as *cloned* code. In this paper, two or more code fragments (i.e., functions) are considered to be clones if they are identical or very similar.

There have been many publications proposing various ways of identifying cloned components in a software system [19, 12, 6, 14, 18, 2]. However, few papers have studied clones dynamic across several versions of the same software system. As a software system evolves, new code fragments may be added, certain parts deleted, modified and/or remain unchanged thus clone *percentage* may change.

Clones evolution across subsequent versions of the same software system can be modeled using time series. A time series is a collection of measures, also said observations, made sequentially in time; examples occur in a variety of fields, ranging from economics to engineering. One of the more interesting outcome when analyzing time series is prediction: given an observed time series it is possible to predict its future values. The prediction requires the identification of a model which adequately describes the observed time series.

Given a set of subsequent versions of a software system, the average number of clones per function in each version can be thought of as a time series; thus a predictive model may be identified.

In this paper a method for monitoring and predicting clones evolution, more precisely the average number of clones per function, across subsequent versions of the same software system is presented. Predicting the future values of a time series is an important issue in many areas: economics, production planning, sales forecasting, stock control.

Monitoring and predicting clones evolution can be used

```

int mysqlCreateDB(sock,DB)
int sock;
char *DB;
{
char *cp;

mysqlDebug(MOD_API,"mysqlCreateDB(%d,%s)\n",sock,DB);
sprintf(packet,"%d:%s\n",CREATE_DB,DB);
writePkt(sock);
(void)bzero(packet,PKT_LEN);
if(readPkt(sock) <= 0)
{
closeServer(sock);
strcpy(mysqlErrMsg,SERVER_GONE_ERROR);
return(-1);
}

/*
** Check the result.
*/

if (atoi(packet) == -1)
{
cp = (char *)index(packet,':');
if (cp)
{
strcpy(mysqlErrMsg,cp+1);
chopError();
}
else
{
strcpy(mysqlErrMsg,UNKNOWN_ERROR);
}
return(-1);
}
return(0);
}

int mysqlDropDB(sock,DB)
int sock;
char *DB;
{
char *cp;

mysqlDebug(MOD_API,"mysqlDropDB(%d,%s)\n",sock,DB);
sprintf(packet,"%d:%s\n",DROP_DB,DB);
writePkt(sock);
(void)bzero(packet,PKT_LEN);
if(readPkt(sock) <= 0)
{
closeServer(sock);
strcpy(mysqlErrMsg,SERVER_GONE_ERROR);
return(-1);
}

/*
** Check the result.
*/

if (atoi(packet) == -1)
{
cp = (char *)index(packet,':');
if (cp)
{
strcpy(mysqlErrMsg,cp+1);
chopError();
}
else
{
strcpy(mysqlErrMsg,UNKNOWN_ERROR);
}
return(-1);
}
return(0);
}

```

Figure 1. mSQL 1.0.6 Clone example.

at least in two different applicative scenarios within the software maintenance field. In particular,

1. the actual effort to evolve and maintain a system is likely to vary depending on the amount of clones found in a system. Thus, the ability to predict clones evolution can be used to define the effort required by the future maintenance activities;
2. a discrepancy between the actual values and the predicted ones is a parameter that may give an *a-posteriori* validation of a certain evolution hypothesis.

The proposed method has been applied to 27 subsequent versions of MiniSQL (mSQL); mSQL is a relational database system developed in C, it is distributed by Hughes Technologies (<http://www.hughes.com.au>). The time span period of the analyzed mSQL versions covers four years, from May 1995 (mSQL 1.0.6) to May 1999 (mSQL 2.0.10). A cross validation procedure was used to measure the method performance.

The rest of the paper is organized as follows. First, background notions on clone detection and time series are summarized in Section 2 for sake of completeness. Then, Section 3 introduces the method for monitoring and predicting clones evolution. Section 4 presents the case study, followed by the experimental results in Section 5. Section 6 compares the present work with previous contributions

while Section 8 summarizes lessons learned and outlines foreseeable research directions.

## 2. Background Notions

The proposed method relies on two basic techniques: clones detection and time series modeling; such techniques are briefly summarized in the following Subsections for sake of completeness.

### 2.1 Clones

In [18], each function in a software system is characterized by its name together with 21 metrics extracted with Datrix software metrics tool (available at <http://www.iro.umontreal.ca/labs/gelo/datrix>). Then, any pair of functions can be compared by these characteristics to yield a rating scored on an ordinal scale. In this paper the focus is on the presence of duplicated or nearly duplicated code so that the exact metrics identity is required to classify two functions as clones. The assumption corresponds to *ExactCopy* and *DistinctName* classes presented in [18].

When studying software system evolution, function identity is usually disregarded in favor of a different concept: clones clusters. A clones cluster can be seen as a set of *similar* code fragments which comprehends identical fragments or fragments exhibiting negligible difference, if any,

from a given fragment prototype. Again the term difference is used with reference to a set of software metrics while the exact definition of *negligible* may depend on the specific analysis goal (e.g., remove duplication or re-modularize the software).

Figure 1 shows an example of nearly duplicated code extracted from mSQL 1.0.6. The functions `mysqlCreateDB` and `mysqlDropDB` differ only in the `sprintf` parameter; in other words, differences could be easily factored out leading to a more abstract functionality implementing both database creation and database destruction.

## 2.2 Time Series

A time series is a collection of observations made sequentially in time. Time series can be modeled using *stochastic processes* [21]. A stochastic process can be described as a statistical phenomenon that evolves in time according to probabilistic laws. Mathematically, it may be defined as a collection of random variables ordered in time and defined at a set of time points which may be continuous or discrete.

One of the possible objectives in analyzing time series is *prediction*: given an observed time series, one may want to predict its future values. The prediction of future values requires the identification of a model describing the time series dynamic. There are many classes of time series models to choose from; the more general is the ARIMA class which includes as special cases the AR, MA and ARMA classes.

A discrete-time process is a purely random process if it consists of a sequence of random variables  $\{Z_t\}$  which are mutually independent and identically distributed. By definition, it follows that purely random processes have constant mean and variance.

Suppose that  $\{Z_t\}$  is a discrete purely random process with mean zero and variance  $\sigma_Z^2$ , then a process  $\{X_t\}$  is said to be a moving average process of order  $q$  (MA( $q$ )) if

$$X_t = Z_t + \beta_1 Z_{t-1} + \dots + \beta_q Z_{t-q} \quad (1)$$

where  $\{\beta_i\}$  are constants. Once the backward shift operator  $B$  is defined as

$$B^j X_t = X_{t-j} \quad (2)$$

a moving average process can be written as

$$X_t = \Theta(B) Z_t \quad (3)$$

where

$$\Theta(B) = 1 + \beta_1 B + \dots + \beta_q B^q \quad (4)$$

Suppose that  $\{Z_t\}$  is a discrete purely random process with mean zero and variance  $\sigma_Z^2$ , then a process  $\{X_t\}$  is said to be an autoregressive process of order  $p$  (AR( $p$ )) if

$$X_t = \alpha_1 X_{t-1} + \dots + \alpha_p X_{t-p} + Z_t \quad (5)$$

where  $\{\alpha_i\}$  are constants. This is rather like a multiple regression model, but  $\{X_t\}$  is not regressed on independent variables but on past variables of  $\{X_t\}$ .

Broadly speaking, a MA( $q$ ) explains the present as the mixture of  $q$  random impulses, while an AR( $p$ ) process builds the present in terms of the past  $p$  events. A useful class of models for time series is obtained by combining MA and AR processes.

A mixed autoregressive moving-average process containing  $p$  AR terms and  $q$  MA terms is said to be an ARMA process of order  $(p, q)$ . It is given by

$$X_t = \alpha_1 X_{t-1} + \dots + \alpha_p X_{t-p} + Z_t + \beta_1 Z_{t-1} + \dots + \beta_q Z_{t-q} \quad (6)$$

where  $X(t)$  is the original series and  $Z(t)$  is a series of unknown random errors which are assumed to follow the normal probability distribution.

Using the backward shift operator  $B$ , the previous equation may be written in the form

$$\Phi(B) X_t = \Theta(B) Z_t \quad (7)$$

where

$$\begin{aligned} \Phi(B) &= 1 - \alpha_1 B - \dots - \alpha_p B^p \\ \Theta(B) &= 1 + \beta_1 B + \dots + \beta_q B^q \end{aligned} \quad (8)$$

A time series is said to be strictly stationary if the joint distribution of  $X(t_1) \dots X(t_n)$  is the same as the joint distribution of  $X(t_1 + \tau) \dots X(t_n + \tau)$  for all  $t_1, \dots, t_n, \tau$ . In other words, shifting the time origin by an amount  $\tau$  has no effects on the joint distributions, which must therefore depend on the intervals between  $t_1, t_2, \dots, t_n$ .

The importance of ARMA processes lies in the fact that a stationary time series may often be described by an ARMA model involving fewer parameters than a pure MA or AR process [8]. However, even if a stationary time series can be efficiently fitted by an ARMA process [22], most time series are non-stationary.

Box and Jenkins introduced a generalization of ARMA processes to deal with the modeling of non-stationary time series [4]. In particular, if in equation (7)  $X_t$  is replaced by  $\nabla^d X_t$ , it is possible to describe certain types of non-stationarity time series. Such a model is called ARIMA (Auto Regressive Integrated Moving Average) because the stationary model is fitted to the differenced data has to be summed or *integrated* to provide a model for the non-stationary data. Writing

$$W_t = \nabla^d X_t = (1 - B)^d X_t \quad (9)$$

the general process ARIMA( $p, d, q$ ) is of the form

$$W_t = \alpha_1 W_{t-1} + \dots + \alpha_p W_{t-p} + Z_t + \dots + \beta_q Z_{t-q} \quad (10)$$

More details can be found in [4].

### 3. The Method

The proposed method for monitoring and predicting the average number of clones per function requires both a technique able to identify a model which adequately describes the time series and a technique to evaluate the average number of clones per function in a given software system version. In this section the clones identification and time series techniques are described.

#### 3.1 Clones Identification

When a system evolves, functionalities are added, modified and/or removed: thus, it is likely that code distribution varies. Following the approach proposed in [18] code fragments (functions in the following) were compared on the basis of software metrics accounting for the layout, the size, the control flow, the function communication and coupling. More precisely, functions were compared on the basis of the following software metrics:

- the number of passed parameters;
- the number of LOC;
- the cyclomatic complexity;
- the number of used/defined local variables;
- the number used/defined non-local variables.

Contrary to previous approaches, e.g., [18], function name and file/unit name were not considered. Indeed, function *identity* may lead to consider different two functions that are in different files of a given version or a function that *migrates* from a file into a different one in a subsequent version. In other words, functions with different names appearing in the same or different files and exhibiting the same values of software metrics were considered clones tout-court without any further subclassification.

function name and file name were not considered, this is because, passing from a version to another, = a function can migrate from one file to a different one or also = developers can simply change its name.

Let  $\mathbf{M}_f = \langle m_1, \dots, m_k \rangle$  be the tuple of metrics characterizing a function  $f$ , where each  $m_i$  is the  $i$ -th software metric chosen to describe  $f$  (e.g., number of passed parameters, number of LOC, cyclomatic complexity, number of used/defined local variables, and number used/defined non-local variables). Theoretically speaking, different sub-systems, or functions could be represented by different collection of metrics, thus discriminating among different features.

For any given function  $f$ ,  $C_f$  the  $f$  clone cluster is the subset of function  $g$ , belonging to the considered software

version  $R_k$ , that exhibit software metric values  $m_i(g)$  identical or *similar* with  $m_i(f)$ :

$$C_f \stackrel{\text{def}}{=} \{g | g \in R_k \wedge m_i(f) \bowtie m_i(g) \wedge m_i \in \mathbf{M}_f\}$$

These represent necessary conditions: the  $\bowtie$  was used to state that  $g$  metrics values,  $m_i(g)$ , may be chosen to meet the specific goal. In order to collect exact, or nearly exact, function duplicates  $\bowtie$  is implemented by the equality while to identify *similar* functions a threshold may be adopted:

$$m_i(f) \bowtie m_i(g) \Rightarrow m_i(f) \leq m_i(g) \leq \theta_u(i) \parallel \theta_l(i) \leq m_i(g) \leq m_i(f) \wedge m_i \in \mathbf{M}_f$$

where  $\theta_l(i)$  and  $\theta_u(i)$  are the lower/upper bounds; inside the range of values  $g$  is considered a clone of  $f$ .

The cluster collection  $C_{R_k}$  for the given version  $R_k$  allows to assess the snapshot over all code duplication structures. A coarser view can be obtained by considering the average cluster size (i.e., the average number of clones per function) and its evolution which gives an overall indication of the number of duplicated function in the snapshot:

$$|C|_{av} = \frac{1}{|C_{R_k}|} \sum_f |C_f|$$

where  $|C_{R_k}|$  is the class cardinality i.e., the number of function in the  $k$  version.

#### 3.2 Time Series Modeling

A wide variety of prediction procedures are available [5, 10, 28]; the method here proposed relies on Box and Jenkins [4] which is quite general: an ARIMA(p,d,q) process includes as special case an ARMA process (i.e., ARIMA(p,0,q) = ARMA(p, q)).

When modeling a time series, attention should be drawn to assess whether the time series is stationary or not. If a time series is stationary it can be modeled through an ARMA(p,q) process, otherwise an ARIMA(p,d,q) is required. A *non-stationary* time series can be described as a time series whose characteristic parameters change over time. Different measures of stationarity can be employed to decide whether a process (i.e., a time series) is stationary or not. In practice, confirming that a given time series is stationary is a very difficult task, unless a closed-form expression of the underlying time series is known. Non-stationarity detection can be reduced to identifying two distinct data segments that have significantly different statistical distributions. Several tests can be used to decide whether

two distributions are statistically different: Student's t-test, F-test, chi-square test and Kolmogorov-Smirnov test [21].

The Box and Jenkins procedure requires 3 main steps briefly sketched below.

1. *Model identification.* The observed time series has to be analyzed to select an ARIMA(p,d,q) process that appears to be the most appropriate; this requires the identification of the  $p, d, q$  parameters.
2. *Estimation.* The actual time series has to be modeled using the ARIMA(p,d,q) process previously defined; this requires the estimation of the  $\{\alpha_i\}$  and  $\{\beta_j\}$  coefficients defined by the equations (8).
3. *Diagnostic Checking.* The residuals (i.e., the difference between the predicted and the actual values) have to be analyzed to verify if the identified model is adequate.

With the *Model Identification Step* the  $d$  value has to be set taking into account whether the time series is stationary (i.e.,  $d = 0$ ) or not (i.e.,  $d > 0$ ). On the other hand, the identification of  $(p, q)$  parameters can be obtained following the *Akaike Information Criterion (AIC)*: the model with smallest AIC has to be chosen [26]. In the *Estimation* step, after the estimation of the  $\{\alpha_i\}$  and  $\{\beta_j\}$  coefficients, it is possible to predict time series future values.

Finally, in the *Diagnostic Checking* step, the model adequacy can be tested plotting the residuals: the residuals of a good model have to be small and random [8].

#### 4. Case Study

Mini SQL (mSQL) is a relational database system; it is developed in C and distributed by Hughes Technologies (<http://www.hughes.com/au>). mSQL relevant features evolution are summarized in Figure 2. In particular, Figure 2 reports the evolution in terms of the number of procedures (i.e., *C functions*) and KLOC of the 27 subsequent versions<sup>1</sup> of mSQL. The first point on the x-axis is related to mSQL 1.0.6 while the last point is concerned with mSQL 2.0.10.

mSQL offers a subset of SQL and its query interface. The first generation product (i.e., mSQL 1.0) was designed to provide high speed access to small data sets using very few system resources on an average UNIX workstation.

Several versions of mSQL have been available in various forms since June 1994 and has undergone many enhancements becoming a very popular and a stable database system for small databases. mSQL 2 is a major redesign, as it can be easily deduced by Figure 2: the size and the number of functions were doubled; moreover, new *builds* (i.e., executables) were added.

<sup>1</sup>The word version is used according to the IEEE Standard Glossary of Software Engineering Terminology (610.12-1990)

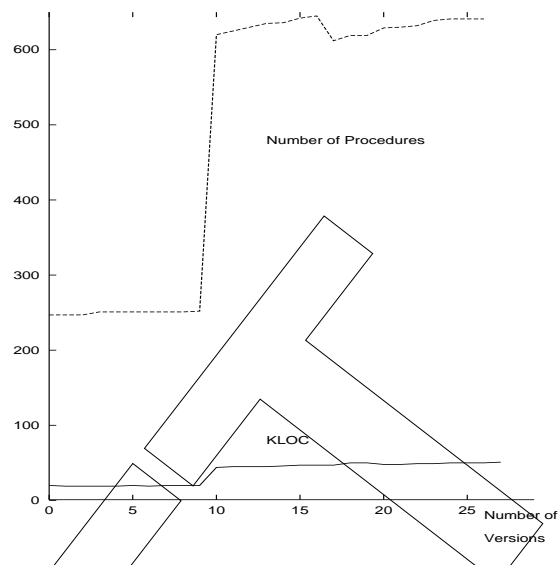


Figure 2. mSQL KLOC and Procedure Number Evolution.

The new implementation goes beyond the initial design goals of mSQL 1.0 and provides functionality suited to larger applications. Moreover, the mSQL 2.0 server has been redesigned to execute multiple queries at the same time. Finally, executables in the distribution grew from 8 to 12: mSQL 2.0 has new functionalities to import/export data and is endowed with W3-mSQL 2.0, the second generation WWW interface package (i.e., w3auth, w3msql, lite). The new W3-mSQL code provides a complete scripting language, with full access to the mSQL API, within an HTML tag.

#### 5. Experimental Results

Several experiments were run on 27 subsequent versions of mSQL (from mSQL 1.0.6 up to mSQL 2.0.10 to empirically measure and quantify the performances of the presented method.

The average number of clones per function was evaluated for each mSQL analyzed version according to the clones detection technique previously described in Subsection 3.1. Such data, shown in the row labelled *Actual* of Table 1 and Table 2, can be thought of as a collection of observations made sequentially in time, and thus modeled using time series.

The experimental activity followed a cross validation procedure [25]; in particular 25 experiments ( $Exp_1, Exp_2, \dots, Exp_{25}$ ) were run: in each experiment a training time series was extracted from the observed time

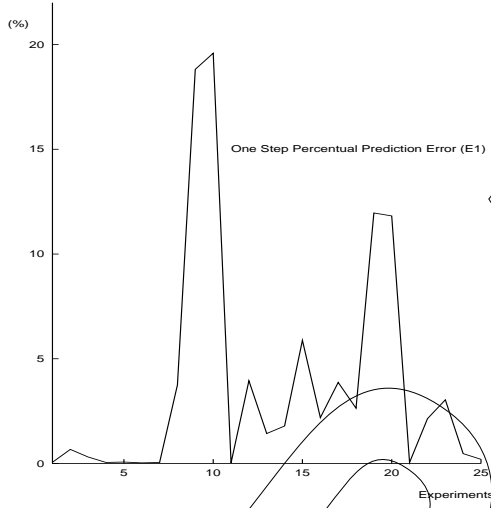
	mSQL 1.0.x									
	6	7	8	9	10	11	12	13	14	16
Ident.	$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$	$x_9$
Actual	1.129	1.127	1.135	1.122	1.130	1.122	1.123	1.121	1.130	1.168
E1 (%)	-	-	0.60	0.67	0.31	0.05	0.07	0.03	0.05	3.75

**Table 1. Actual Time Series Values and Percent Prediction Errors (mSQL 1.0.x)**

	mSQL 2.0.x																
	B1	B2	B3	B4	B5	B6	B7.1	1	2	3	4	5	6	7	8	9	10.1
Ident.	$x_{10}$	$x_{11}$	$x_{12}$	$x_{13}$	$x_{14}$	$x_{15}$	$x_{16}$	$x_{17}$	$x_{18}$	$x_{19}$	$x_{20}$	$x_{21}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{25}$	$x_{26}$
Actual	1.44	1.43	1.43	1.38	1.36	1.33	1.41	1.45	1.40	1.43	1.28	1.45	1.40	1.41	1.44	1.43	1.44
E1 (%)	18.82	19.59	0.03	3.95	1.43	1.79	5.88	2.19	3.87	2.64	11.96	11.82	0.05	2.14	3.04	0.47	0.20

**Table 2. Actual Time Series Values and Percent Prediction Errors (mSQL 2.0.x)**

series related to the average number of clones per function; then, the training time series was analyzed and modeled to predict its future values; finally the predicted values were matched against the actual values and the method performance was assessed in terms of *percent prediction error*.



**Figure 3. Percent Prediction Errors.**

Given the observed time series related to average number of clones per function, the  $k^{th}$  experiment ( $1 \leq k \leq 25$ ) was run on the  $k^{th}$  training time series ( $tts_k$ ) defined as follows

$$tts_k = \{x_0, \dots, x_{k+1}\} \quad (11)$$

where  $x_0$  is the average number of clones per function in mSQL 1.0.6,  $x_1$  is the average number of clones per function in mSQL 1.0.7 and so on (cfr. the row labelled *Ident.* in Table 1 and Table 2).

During the  $k^{th}$  experiment (i.e.,  $E_k$ ) the one step ahead value (i.e.,  $\hat{x}_{k+2}$ ) was predicted, the predicted value was compared against the actual one (i.e.,  $x_{k+2}$ ) in order to evaluate the one step ahead percent prediction error.

If  $\hat{x}_T$  is the predicted value and  $x_T$  is the actual value, the percent prediction error is defined as follows:

$$\text{percent prediction error} = \frac{\text{abs}(x_T - \hat{x}_T)}{x_T} * 100 \quad (12)$$

The one step ahead percent prediction error related to the performed experiment is showed both in Table 1 and Table 2 in the rows labeled as E1. For example, in the last column of Table 1 the one step ahead percent prediction evaluated during the experiments  $Exp_8$  is reported. A plotting of E1 is also shown in Figure 3.

As previously stated, monitoring and predicting clones evolution can be used at least in two different applicative scenarios and the available data support such a conjecture.

In particular, the actual effort to evolve and maintain a system is likely to vary depending on the amount of clones found in a system. For this perspective, predicting the average number of clones per function may give a significant support in defining the actual effort required to maintain and evolve such a new software system version. It is worth noting that on average the percent one step ahead prediction error was 3.81%: on the entire sequence of the mSQL analyzed versions, clones evolution was predicted with a quite acceptable error.

The analysis of the change logs related to the analyzed mSQL versions highlighted that during the evolution from mSQL 1.0.6 to mSQL 1.0.16, the principal activity was concerned with bug fixing, without significant changes occurred at design and code level. On the other hand, dramatic changes were introduced in mSQL 2.0.B1: old features were redesigned and new functionalities were added. The following  $\beta$  versions (i.e., from mSQL 2.0.B2 up to 2.0.B7.1) were devoted to fixing the new discovered bugs. Also the versions from 2.0.1 to 2.0.3 were mainly concerned with bug fixing while significant changes were introduced with the 2.0.4. In particular, in mSQL 2.0.4 a partial re-

modularization was performed to improve *search* functionalities. Finally, bugs fixing was the principal activity in the remaining analyzed versions (i.e, from mSQL 2.0.5 up to 2.0.10.1) too.

The analysis of the one step ahead percent prediction error curve, shown in Figure 3, highlights two peaks. The first peak appears with the predicted values related to the first two  $\beta$  versions (2.0.B1 and 2.0.B2); the other peak appears with the prediction related to mSQL 2.0.4. According to available data, whenever major changes occurred the predicted values were affected by an error bigger than 11%, while in all the other cases the error is less than 6%. Thus, a discrepancy between the actual and the predicted values is a parameter that may give an *a-posteriori* validation of a certain evolution hypothesis.

For example, whenever a customer should acquire a new version of a software system he/she could be interested in the evaluation of the evolution extent; such evaluation can be supported by matching predicted values with the actual values of the average number of clones per function. In other words, when a new version which is supposed to implement dramatic changes is delivered, the discrepancy between the predicted and actual values may be an indicator of the real extent of the changes.

## 6. Related Work

Previous research has studied both the detection and the use of clones for widely varying purposes including program comprehension, documentation, quality evaluation or system and process restructuring. Several techniques have been investigated in the literature for the detection of clones in software systems. Some techniques are based on a full text view of the source code [12, 1]. Other approaches, such as those pursued by Mayrand et al. [18] and Kontogiannis et al. [14] focus on whole sequence of instructions (BEGIN-END blocs or functions) and allow the detection of similar blocks using metrics. Moreover, Kontogiannis et al. [14] detect clones using two other pattern matching techniques, namely dynamic programming matching and statistical matching between abstract code descriptions patterns and source code. Finally, another clone detection technique relies on the comparison of subtrees from the AST (Abstract syntax tree) of a system. Baxter et al. [2] have investigated this technique.

Several applications of clone detection have also been investigated: Johnson [12] visualizes redundant substrings to ease the task of comprehending large legacy systems; Mayrand et al. [18], as well as Lagie et al. [15], document the cloning phenomenon for the purpose of evaluating the quality of software systems; Lagie et al. [15] have also evaluated the benefits in terms of maintenance of the detection of cloned methods; finally, Baxter et al. [2] restructure sys-

tems by replacing clones with macros to reduce the quantity of source code and facilitate maintenance.

The present work is not focused on clones detection technology but rather on the application of the clones detection to software system evolution. Moreover, function *identity* was not considered and this allowed us to overcome the problems highlighted in [15].

The problem of estimating the effort of the software projects has long been recognized as a key to successful software management and several authors have proposed estimation methods and tools [3, 27, 11, 24, 13, 9, 20, 7].

The method presented in this paper is complementary, as it identifies clones evolution as a parameter which may help both to estimate the future maintenance effort and validate evolution hypothesis.

## 7. Conclusions

In this paper a method for monitoring and predicting the evolution of clones across subsequent versions of a software system has been presented.

An experimental activity has been performed on 27 subsequent versions of mSQL to empirically quantify the performance of the method. The experiments followed the cross validation schema and simulated a real use of the method.

Preliminary results are encouraging: on average an error of 3.81% has been observed.

The monitoring and prediction of clones evolution may be employed at least in two different applicative scenarios: the prediction may be used to define the actual effort required to maintain and evolve future software system versions. Moreover, the analysis of the mSQL change logs highlighted that a discrepancy between the estimated and the actual values was due to dramatic changes in the system. Thus, the discrepancy between the actual and the predicted values is a parameter that may give an *a-posteriori* validation of a certain evolution hypothesis.

This paper makes two main contributions: it proposes time series modeling as an applicable approach within the software maintenance field and gives an insight of clones evolution across subsequent versions of the same software system. In fact, even if there have been many publications concerned with the techniques aimed at identifying cloned components in a software system, few papers have studied clone dynamics across several versions of the same software system.

Further work will be devoted to assess the performance of the method on different software systems.

## References

- [1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Working Conference on Reverse Engineering*, pages 86–95, July 1995.
- [2] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 368–377, 1998.
- [3] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [4] G. Box and M. Jenkins. *Time Series Forecasting Analysis and Control*. Holden Day, San Francisco (USA), 1970.
- [5] R. G. Brown. *Smoothing, Forecasting and Prediction*. Prentice Hall, 1963.
- [6] E. Buss, R. D. Mori, W. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Muller, J. M. S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster, and K. Wong. Investigating reverse engineering technologies for the cas program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.
- [7] G. Caldiera, G. Antoniol, R. Fiutem, and C. Lokan. A definition and experimental evaluation of function points for object-oriented systems. In *Proc. of the Fifth International Symposium on Software Metrics - METRICS98*, pages 167–178, Nov 2-5 1998.
- [8] C. Chatfield. *The Analysis of the Time Series*. Chapman & Hall, 1996.
- [9] J. C. Granja-Alvarez and M. J. Barraco-Garcia’. A method for estimating maintenance cost in a software project: a case study. *Software Maintenance Research and Practice*, 9:161–175, 1997.
- [10] A. Harvey. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1989.
- [11] IFPUG. *Function Point Counting Practices Manual, Release 4.0*. International Function Point Users Group, Westerville, Ohio, 1994.
- [12] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183, October 1993.
- [13] M. Jorgensen. Experience with the accuracy of software maintenance task effort prediction models. *IEEE Transactions on Software Engineering*, 21(8):674–681, 1996.
- [14] K. Kontogiannis, R. D. Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:77–108, March 1996.
- [15] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 314–321, 1997.
- [16] M. M. Lehman and L. A. Belady. *Software Evolution - Processes of Software Change*. Academic Press, London, 1985.
- [17] M. M. Lehman, D. E. Perry, and J. F. Ramil. On evidence supporting the feast hypothesis and the laws of software evolution. In *Proc. of the Fifth International Symposium on Software Metrics*, pages 84–88, Bethesda, Maryland, November 1998.
- [18] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 244–253, Monterey, CA, Nov 1996.
- [19] T. J. McCabe. Reverse engineering, reusability, redundancy: the connection. *American Programmer*, 3:8–13, October 1990.
- [20] A. Minkiewicz. Measuring object-oriented software with predictive object points. In *Proceedings of 8<sup>th</sup> European Software Control and Metrics Conference*, Atlanta, May 1997.
- [21] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, 1984.
- [22] D. Piccolo and C. Vitale. *Metodi Statistici per l’Analisi Economica*. Il Mulino, 1989.
- [23] H. Sneed. Estimating the costs of object-oriented software. In *Proceedings of Software Cost Estimation Seminar*, 1995.
- [24] H. M. Sneed. Estimating the cost of software maintenance tasks. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 168–180, Opio, Nice, Oct. 1995. IEEE Press.
- [25] M. Stone. Cross-validators choice and assesment of statistical predictions (with discussion). *Journal of the Royal Statistical Society B*, 36:111–147, 1974.
- [26] W. Venables and B. Ripley. *Modern Applied Statistic with S-PLUS*. Springer, 1999.
- [27] F. Wellman. *Software Costing*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [28] M. West and P. Harrison. *Bayesian Forecasting and Dynamic Models*. Springer-Verlag, 1989.