Integrate the GM(1,1) and Verhulst Models to Predict Software Stage Effort

Yong Wang1, Qinbao Song1, Stephen MacDonell2, Martin Shepperd1, and Junyi Shen1

1Department of Computer Science and Technology, Xi’an Jiaotong University, Xi’an 710049, China
markwy@126.com, qbsong@mail.xjtu.edu.cn, jyshen@mail.xjtu.edu.cn

2SERL, Auckland University of Technology
Private Bag 92006, Auckland 1142, New Zealand
stephen.macdonell@aut.ac.nz

3Empirical Software Engineering Research Group,
School of Design, Engineering and Computing,
Bournemouth University, Bournemouth BH13LT, UK
mshepper@bmth.ac.uk

Abstract

Background: Software effort prediction clearly plays a crucial role in software project management. Problem: In keeping with more dynamic approaches to software development, it is not sufficient to only predict the whole-project effort at an early stage. Rather, the project manager must also dynamically predict the effort of different stages or activities during the software development process. This can assist the project manager to reestimate effort and adjust the project plan, thus avoiding effort or schedule overruns. Method: This paper presents a method for software physical time stage-effort prediction based on grey models GM(1,1) and Verhulst. This method establishes models dynamically according to particular types of stage-effort sequences, and can adapt to particular development methodologies automatically by using a novel grey feedback mechanism. Result: We evaluate the proposed method with a large-scale real-world software engineering dataset, and compare it with the linear regression method and the Kalman filter method, revealing that accuracy has been improved by at least 28% and 50%, respectively. Conclusion: The results indicate that the method can be effective and has considerable potential. We believe that stage predictions could be a useful complement to whole-project effort prediction methods.

Index Terms - Grey prediction, software project management, software project stage-effort prediction.

1. INTRODUCTION

ACCURATE and unbiased software effort prediction is clearly important in terms of enabling software developers/managers to make a reasonable bid or form a plan of activities—consequently, an extensive body of research has addressed this facet of project management (see, for example, [1], [4], [5], and [8]–[11]). On the other hand, the capability to predict the effort required in different stages during the software development process is also important but rather less studied [3], [10], [12]–[16]. Such a capability enables (or at least should enable) project managers to identify potential effort overrun risk during the project and reallocate resources when necessary. It has been estimated that around 75% of all projects overrun their schedules due to inaccurate effort prediction [5]. The development and use of software stage-effort prediction methods have the potential to ensure that predictions are revisited and revised on an ongoing basis, and the result being that the accuracy of predictions should improve.

This is not to say, however, that the accurate prediction of software stage effort is straightforward. The software development process can proceed in a sporadic manner despite the best-laid plans. It is influenced by many uncertain and challenging-to-measure factors, such as individuals’ levels of expertise, project difficulty, and technical complexity [3], [14], [17], [18]. Moreover, software development is largely a continuous and cumulative process. The work of prior stages forms the basis of current and subsequent stages, and there is evidence to suggest that there are some inherent relationships between the effort of prior and subsequent stages [3], [10], [12], [13]—although the nature of those relationships is not yet clear. This lends motivation to the idea of using prior stage effort to predict subsequent stage effort, a prediction process that should be continuous and dynamic in line with contemporary approaches to software development.

The stage-effort prediction problem has been addressed...
using a variety of data analysis methods (see Section 2)—among them, linear regression (LR) analysis has been popular. In some cases, however, the relationships between different process stages was found to be not particularly strong, leading to some instances of large prediction error. In particular, predictions made in the early stages of a project tend to be more challenging—later-stage predictions can leverage the greater certainty that accrues with progress. Early-stage prediction must be performed in the context of data starvation, a context that is not conducive to commonly employed statistical methods (such as regression and those based on time series) and machine learning methods that usually require large data samples to determine statistical features of the series to build prediction models.

Grey system theory (GST), a system engineering theory based on the uncertainty of small samples, was first proposed by Deng in 1982 [19]. In keeping with the techniques described before, i.e., it can enable the establishment of a prediction model using just a small amount of known data. In the context of data starvation, GST is known to be effective and has been widely applied to address real-world problems in the domains of energy management [20], mobile communication [21], instrument measurement [22], stock price analysis [23], and image processing [24], [25]. In addition, Song et al. [26] used GST to address the whole-project effort prediction problem in software engineering. They used grey relation analysis (GRA) derived from GST to select more effective feature subsets and similar projects to support a prediction process. Their results showed that GRA can resolve the effort prediction problem with high prediction accuracy. Encouraged by this successful work, we now explore the stage-effort prediction problem using GST in this research.

As we know, there are many project life-cycle models used in the software development domain such as iterative cycles, traditional waterfall models, and so forth. These, in turn, are split into different phases, e.g., feasibility, high-level design, etc. Each of these phases can have complex and, on occasions, ill-defined mappings to physical time units (such as week, month, and quarter). These time units have most generality and are clearly very important in project scheduling. For this reason, in this paper, we focus on software physical time unit, which, for clarity, we refer to as stage-effort prediction. This is in contrast to prediction based on phases, as shown by MacDonell and Shepperd [12], that used a seven-phase classification scheme.

We propose a method named GV (GM(1,1) and Verhulst). GV takes full advantage of two grey models of GST - GM(1,1) and Verhulst - to predict future stage effort in light of the law derived from records of prior stage effort, and it can adapt to particular development methodologies automatically by using a novel grey feedback (GFB) mechanism. We validate the method on a large-scale software engineering data repository and obtain very promising results.

The remainder of this paper is organized as follows. In the next section, we discuss related work on, and then present the basic concepts and the prediction principles of the grey models, followed by the introduction of the proposed stage-effort prediction method GV and an example. After that, we describe the datasets and the experimental method we used. The results are then described and discussed, with a concluding discussion presented in the last section.

2. RELATED WORK

Surprisingly, the empirical study of software stage-effort prediction during the development process has received limited attention. The authors are aware of only four studies that have investigated this issue empirically (one being a previous study involving two of the authors [12]). Kulkarni et al. [13] employed a form of transformation matrix, referred to as a minimodel, to predict effort for each life-cycle phase. By combining the set of distinct-phase minimodels into a single overall model, it is possible to determine the output measures from the final phase using input measures to the first phase. This method was applied to a military project by Kulkarni et al., and while it appeared to be potentially effective, prediction accuracy data was not presented. Ohlsson and Wohlin [24] used artifact-based proxies of project scope to augment existing within-project prediction and planning processes. They concluded that the approach has potential in making plans, and deviations from those plans, which are more visible, and that it can increase managers’ confidence that predictions are of the right order. MacDonell and Shepperd [12] predicted life-cycle phase effort for 16 software projects using a simple LR method. They used prior-phase effort data to predict the effort needed for subsequent phases in each of the projects. The results showed that the method produced better predictions than those provided by the project managers. A study reported by Abrahamsson et al. [27] utilized regression and neural network methods to generate iterative prediction and planning models suitable for projects developed using agile methods. They concluded that the approach improves prediction when compared to whole-project efforts, and that such predictions are stable and convergent, attributes that are essential in terms of effective planning.

A variety of techniques, model inputs, and stage units were used in the studies just described. Compared with these studies, our method – GV – has the following differences or characteristics.

1) GV is capable of predicting project effort for physical time stages. The other studies depend on the use of the life-cycle phase [12], [13], [15] or the iterative development cycle [27].

2) GV uses the inherent trend embedded in the prior stage effort to predict. It does not need the historical data of outputs to build the model. The other studies focused on utilizing the relationships between input and output measures to build the models. MacDonell and Shepperd [12] examined the correlations between
phases (e.g., between design and implementation), Ohlsson and Wohlin [15] mapped artifact-based proxies to effort, Abrahamsson et al. [27] used LR and neural network methods to express the relationships between inputs and outputs, and Kulkarni et al. [13] also constructed a matrix-based model to reflect relationships between inputs and outputs.

3) GV requires project effort itself only as input. MacDonell and Shepperd [12] also used project effort as input. Abrahamsson et al. [27] used several estimated predictor variables and the effort of previous iterations as inputs. Ohlsson and Wohlin [15] used their artifact-based proxies as inputs, and Kulkarni et al. [13] used object measures (e.g., source lines of code, Ada packages, and data flows).

In short, GV is a novel method to address the problem of stage-effort prediction in software; in principle, however, it would appear to have potential. We now provide an introduction to grey models and to the GV approach.

### 3. GREY MODEL

GST uses grey models to make predictions. In this section, we will introduce the basic concepts and two grey models used in this paper, respectively.

#### A. Basis

Generally, the nonnegative raw data sequence has no obvious patterns in uncertain circumstances [19], and it is difficult to find a proper curve to fit it, but after an accumulated generating operation (AGO), the generated sequence will monotonically increase and reflect a strong exponential character. We call this the “accumulated generating grey exponential law.” Therefore, we can find an optimum exponential curve to simulate it. The AGO is defined as follows.

Let $X^{(0)}$ be the original nonnegative sequence, the AGO sequence $X^{(1)}$ can be generated as

$$X^{(1)}(k) = \sum_{i=1}^{k} X^{(0)}(i), k = 1, 2, \ldots, n. \quad (1)$$

For example, let $X^{(0)} = (3, 5, 4, 7)$, then $X^{(1)} = (3, 8, 12, 19)$. If the generating exponential character is not obvious, further AGOs may be applied. For returning the data to the original condition, the inverse AGO (IAGO) will be applied. Therefore, AGO and IAGO are a pair of inverse sequence operators. The operation of IAGO is defined as follows:

$$x^{(0)}(1) = x^{(1)}(1)$$

$$x^{(0)}(k + 1) = x^{(1)}(k + 1) - x^{(1)}(k), k = 1, 2, \ldots, n. \quad (2)$$

In GST, the curves used to fit AGO sequences can be represented by differential equations. We refer to these differential equations as grey models. There are many kinds of grey models available according to the different kinds of AGO sequences, such as GM(1,1), Verhulst, DGM, etc. Among them, the GM(1,1) is commonly used to simulate exponential-type sequences; therefore, it is suitable to describe any monotonic increasing procedure. Verhulst is suggested to simulate sequences with saturated trend. The typical curves of these two models [28] are illustrated in Fig. 1, where $k$ denotes the sequence element index and $x^{(0)}(k)$ denotes the sequence element value.

#### B. GM(1,1) Model

A form of single variable and first-order linear dynamic differential equation

$$\frac{dX^{(1)}}{dt} + aX^{(1)} = b \quad (3)$$

where $X^{(1)}$ is the AGO sequence of the original sequence, the parameters $a$ and $b$ are called the development coefficient and grey action quantity, respectively, which can be obtained from the following expression:

$$\hat{\alpha} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = (B^T B)^{-1} B^T Y_N \quad (4)$$

where

$$B = \begin{bmatrix} -z^{(1)}(2) & 1 \\ -z^{(1)}(3) & 1 \\ \vdots & \vdots \\ -z^{(1)}(n) & 1 \end{bmatrix} \quad (5)$$

$$Y_N = \begin{bmatrix} x^{(0)}(2) \\ x^{(0)}(3) \\ \vdots \\ x^{(0)}(n) \end{bmatrix} \quad (6)$$

and

$$z^{(1)}(k) = \frac{x^{(1)}(k) + x^{(1)}(k-1)}{2}, k = 2, 3, \ldots, n. \quad (7)$$

Then, the time response sequence of GM(1,1) can be written as follows:

$$z^{(1)}(k+1) = \left( x^{(0)}(1) - \frac{\beta}{a} \right) e^{-ak} + \frac{b}{a}, k = 1, 2, \ldots, n. \quad (8)$$
where $\hat{x}$ denotes the prediction of $x$. $\hat{x}^{(0)}(k+1)$ can be obtained by IAGO as
\[
\hat{x}^{(0)}(k+1) = \hat{x}^{(i)}(k+1) - \hat{x}^{(i)}(k), \quad k = 1, 2, ..., n. \quad (9)
\]

### C. Verhulst Model

The Verhulst model is a form of single variable and second-order differential equation
\[
\frac{dX^{(2)}}{dt} + aX^{(i)} = b(X^{(i)})^2 \tag{10}
\]
where $a$ and $b$ can be obtained by
\[
\hat{a} = \left[ \begin{array}{c}
\hat{a}_1 \\
\hat{a}_2 \\
\vdots \\
\hat{a}_n
\end{array} \right] = (B^T B)^{-1} B^T Y_n \tag{11}
\]
where
\[
B = \left[ \begin{array}{cccc}
-z^{(1)}(2) & (z^{(1)}(2))^2 \\
-z^{(1)}(3) & (z^{(1)}(3))^2 \\
\vdots & \vdots \\
-z^{(1)}(n) & (z^{(1)}(n))^2
\end{array} \right] \tag{12}
\]
\[
Y_n = \left[ \begin{array}{c}
x^{(0)}(2) \\
x^{(0)}(3) \\
\vdots \\
x^{(0)}(n)
\end{array} \right] \tag{13}
\]
and
\[
z^{(1)}(k) = \frac{x^{(2)}(k) + (a^{(2)}(k))^2}{2}, \quad k = 2, 3, ..., n. \tag{14}
\]

The time response sequence of Verhulst can be expressed as follows:
\[
\hat{x}^{(i)}(k+1) = \frac{a x^{(1)}(k)}{b x^{(1)}(k)} + \left( \frac{a}{b} \right)^2 x^{(1)}(k) \quad k = 1, 2, ..., n. \tag{15}
\]

Note that, in practice, the saturated sequence already has some degree of exponential form, so the original sequence can be regarded as $X^{(1)}$ directly, this means that we need not apply AGO to $X^{(0)}$ to obtain $X^{(1)}$. And $X^{(0)}$ in (13) should be the IAGO sequence of the original sequence.

For the Verhulst model, the prediction values of the original sequence can be obtained from (15).

### 4. ANALYSIS OF SOFTWARE STAGE-EFFORT SEQUENCES

The characteristics of software stage-effort sequences\(^1\) are influential in determining the use of appropriate prediction methods. In this section, we introduce some general characteristics of software stage-effort sequences and demonstrate some preliminary analysis of subsequences\(^2\) and sequence sets\(^3\).

#### A. Overview

Software stage-effort sequences are records of the software development process. Investigation of stage-effort sequence data used in this study reveals that around 97% of all projects comprise fewer than 24 stages, where a stage represents one month. Therefore, software stage-effort sequences are finite, and generally, quite short in length. In addition, in many cases, software stage-effort sequences do not exhibit a regular shape over the duration of the project, such as the bell-shaped curve or similar, as we might imagine. In fact, they almost have no typical patterns. Fig. 2 provides an illustrative depiction of this scenario, using some of the software stage-effort data analyzed in this study.

From Fig. 2, we note that at least some of the sequences are very irregular; however, because all of the sequences are plotted on this single graph, it is difficult to identify any consistent patterns within or among them. For ease of observation and comment, we pick out two curves at random from Fig. 2 and show them in Fig. 3. We find that the trend over the whole sequence changes frequently. Within segments of the two sequences, however, the trend demonstrates a degree of stability. In the absence of any consistent whole-sequence patterns, it may be that subsequences can be used to reflect changes in stage-ef

\[\text{Effort} \quad \text{(operation man)}\]
\[\text{Stage} \quad \text{(month)}\]

\[1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \]

Fig. 2. Curves of some software stage-effort sequences.

#### B. Subsequences

Usually more recent data points are more important in predicting the next data point, so we use recent stage-effort data points to compose each model subsequence. How many stage-effort observations should we use to build a prediction model? In the early stages of the software development process, the number of stage-effort values available is small. A grey model can establish a prediction model with small data samples, but usually more data is helpful. On the other hand, a model that employs many past observations is not always sensitive to data that are liable to frequent and/or significant changes.

\[\text{Sequence: a series of physical time stage-effort values of a software project.}\]

\[\text{Subsequence: a subset of a software stage-effort sequence for building prediction model.}\]

\[\text{Sequence set: a collection of software stage-effort sequences.}\]

---

\(^1\) Sequence: a series of physical time stage-effort values of a software project.

\(^2\) Subsequence: a subset of a software stage-effort sequence for building prediction model.

\(^3\) Sequence set: a collection of software stage-effort sequences.
or can incorporate obsolete values in terms of their relevance to future predictions. Based on these considerations, we use subsequences consisting of the most recent three stage-effort data values.

![Fig. 3. Stage-effort curves of two projects.](image)

We find that a subsequence with three elements can be classified into one of the four classes according to its trend and shape. Fig. 4 portrays the four classes of subsequences with characteristics abstracted.

![Fig. 4. Four classes of stage-effort subsequences.](image)

Considering the sequence representations in Fig. 4 in relation to the model types referred to in Fig. 1, we note that curves 1 and 2 are similar to the curves of Verhulst and GM(1,1), respectively. It is also evident that curves 1 and 3, and curves 2 and 4 are symmetrical pairs. If we flip the class 3 or 4 curve in an up/down direction, the curve adopts the form of curve 1 or 2, so they can be dealt with as per curve 1 or 2. Therefore, we can use GM(1,1) or Verhulst to build an appropriate prediction model dynamically according to the types of subsequences identified.

Although prior research has failed to find clear laws by observation [29], [30], these works have shown that by using GM(1,1) or Verhulst grey models to make predictions using historical project data grouped by development methodology (or industry, etc.), the mean prediction biases for each group are significantly different. These differences can be regarded as accumulate effects of the particular methodology applying on the predictions. On the other hand, it indicates that the grey models do not differentiate between the methodologies. Adopting a system view, we can regard the stage-effort sequences as inputs, the grey models as a process system, and the biases as outputs. We can then use the mean biases as feedback to adjust the system to obtain more optimal and consistent outputs. The feedback has relations with the development methodologies but its components are not completely clear, so we refer to it as “grey feedback” or GFB. Fig. 5 depicts the mechanism of GFB.

![Fig. 5. GFB mechanism.](image)

5. PROPOSED SOFTWARE STAGE-EFFORT PREDICTION METHOD GV

A. General Method

In general, GV uses subsequences to build the grey models, and uses sequence sets to obtain GFB. The grey models are then used to make predictions, and GFB is acted on in terms of prediction adjustment, leading to the production of an optimal prediction. Therefore, GV leverages both local (within sequence) and global (sequence set) information to produce predictions. Assume that a live project has a stage-effort sequence, \( \{x(1), x(2), \ldots, x(n)\} \) (\( n \geq 3 \)). GV can use the most recent three stage-effort values \( x(n-2), x(n-1), \) and \( x(n) \) to predict the effort of stage \( n+1 \), i.e., \( x(n+1) \). The prediction procedure of GV includes the following steps.

Step 1: Construct the model subsequence. Use the most recent three stage-effort data values \( x(n-2), x(n-1), \) and \( x(n) \) as model subsequence.

Step 2: Smooth the subsequence. Use the modified moving average of order 3 to eliminate unwanted fluctuations of the subsequence (see Section 5-B for details).

Step 3: Determine the subsequence type according to the
trend (up/down) and shape (raised/concave) (see Section 5-C for details).

Step 4: Establish a grey model dynamically according to the subsequence type and produce a preliminary prediction value \( \hat{x}(n+1) \) (see Section 5-D for details).

In order to facilitate the following description, we represent steps 1–4 as a function

\[
\hat{x}(n+1) = \text{Grey\_Predict}(x(n-2), x(n-1), x(n)), n \geq 3.
\]

(16)

where the input to the function is the subsequence and the output is the prediction value of stage effort for the next stage \( n+1 \).

Step 5: Adjust the prediction result using the stage-effort adjustment coefficient (SAC) (see Section 5-E for details), and then obtain the optimal prediction as

\[
\hat{x}_{\text{opt}}(n+1) = \hat{x}(n+1) \cdot \text{SAC}.
\]

(17)

For ease of description, we denote the overall GV prediction procedure (from steps 1–5) by a function

\[
\hat{x}_{\text{opt}}(n+1) = \text{GV}(x(n-2), x(n-1), x(n)), n \geq 3.
\]

(18)

where \( \hat{x}_{\text{opt}}(n+1) \) is the final effort prediction value of stage \( n+1 \). Fig. 6 portrays the details of the general prediction procedure of GV.

**B. Modified Moving Average of Order 3**

Smooth transformation can reduce noise-based fluctuations in the sequences. A common transformation method is the weighted moving average of order \( k \) [31]

\[
\frac{\omega_1 y_1 + \omega_2 y_2 + \cdots + \omega_k y_k}{\omega_1 + \omega_2 + \cdots + \omega_k}, \quad \frac{\omega_2 y_1 + \omega_3 y_2 + \cdots + \omega_k y_{k-1} + y_k}{\omega_2 + \omega_3 + \cdots + \omega_k + 1},
\]

\[
\frac{\omega_3 y_1 + \omega_4 y_2 + \cdots + \omega_k + 2 y_{k+2}}{\omega_3 + \omega_4 + \cdots + \omega_k + 2}, \cdots
\]

(19)

where \( Y = (y_1, y_2, y_3, \ldots) \) is a given sequence and \( \omega_i \) is the weight. After transformation, \( k-2 \) head and tail data are discarded. This would likely raise problems in software stage-effort prediction as, in general, the stage-effort data of a software project are limited, particularly, in the early stages of development. The aforementioned transformation makes small data samples even smaller, perhaps of insufficient size to construct a prediction model. Therefore, we modify (19) by reserving the head and tail data. For example, the weighted moving average of order 3 used in this paper can be treated as

\[
\frac{3y_1 + y_2}{4}, \frac{y_{n-1} + 2y_k + y_{k+1}}{4}, \cdots, \frac{y_{n-1} + 3y_k}{4}.
\]

(20)

Specifically, for a sequence of three elements \( (y_1, y_2, y_3) \), the transformed sequence is

\[
\frac{3y_1 + y_2}{4}, \frac{y_1 + 2y_2 + y_3}{4}, \frac{y_2 + 3y_3}{4}.
\]

(21)

The three elements can form a folding line. The basic shape of the folding line will not change after the transformation, and a straight line will still be a straight line after the transformation. Fig. 7 illustrates the original and transformed sequences where the \( x \)-axis denotes the indexes of the elements and the \( y \)-axis denotes the values. The circles represent the original data and the squares represent the transformed values.

From Fig. 7, we can see that the transformed sequence is more regular than the original but that the original trend is preserved.

![Fig. 6. Prediction procedure of GV.](image)

![Fig. 7. Modified moving average of order 3.](image)

**C. Determination of Subsequence Type**

We assume that the subsequence is composed of \( (y_1, y_2, y_3) \). The type determination procedure includes two stages. Fig. 8 shows in pseudocode the detailed determination procedure of subsequence type.

First, classify the subsequence into up or down classes by comparing \( y_1 \) with \( y_3 \). If \( y_3 > y_1 \), we say that the subsequence has a total up trend, if \( y_3 < y_1 \), we say it has a total down trend, if \( y_3 = y_1 \) (horizontal), we regard it as a straight line (steps 1 and 8, respectively, in Fig. 8).

We then classify the subsequence into raised or concave classes by comparing \( y_2 \) with the median of \( y_1 \) and \( y_3 \), i.e., \( (y_1 + y_3)/2 \): for an up class subsequence, if \( y_2 > (y_1 + y_3)/2 \), we say the subsequence is raised, type is 1 (steps 3 and 4), if \( y_2 < (y_1 + y_3)/2 \), we say it is concave, type is 2. When \( y_2 = (y_1 + y_3)/2 \), it is a straight line, we
include it in the concave class (steps 5 and 6) and 2) for a
down class subsequence (including $y_3=y_1$), if $y_2 < (y_1+y_3)/2$, we say the subsequence is concave, type is 3
(steps 10 and 11), if $y_2 > (y_1+y_3)/2$, we say it is raised,
type is 4. When $y_2=(y_1+y_3)/2$, it is a straight line, we
include it in the raised class (steps 12 and 13).

![Fig. 8. Determination of subsequence type procedure.]

**D. Prediction Model**

GV establishes models dynamically according to the
types of stage-effort subsequence. Considering the
subsequence representations in Fig. 4 in relation to the
model types referred to in Fig. 1, we note that curve 1 is
up-raised, i.e., a saturated sequence, so a Verhulst model
can be used to represent it. Curve 2 is up-concave, i.e., an
exponential sequence, and thus, is theoretically suitable
to be predicted by GM(1,1). Curves 3 and 4 are down
sequences that cannot use GM(1,1) or Verhulst directly,
because GM(1,1) and Verhulst require the sequences to
have a total upward trend for accurate predictions.
Therefore, we must first convert these curves to up-style
sequences. In fact, as noted, it is clear that curves 1 and 3,
and curves 2 and 4 are symmetrical pairs. If we flip the
class 3 or 4 curve in an up/down direction, it adopts the
form of curve 1 or 2. We can then use GM(1,1) or
Verhulst to model and predict the curves. After
predicting, we need to flip again to ensure that the models
match the original sequences. The detailed procedure is
shown in Fig. 9. In summary, the classes and suitable
models for various types of original data subsequences
are listed in Table I.

**E. Obtaining SAC and GFB**

SAC forms a bridge between the grey models and
particular development methodologies. Using it, GV can
fit the particular stage-effort sequences better and obtain
more accurate results. In prior research [29], [30], a bias
correction method has been used, but it tends to overfit.
In this study, we make improvements based on the prior
work, and propose a novel adjustment coefficient SAC as
follows:

$$SAC = \frac{GFB/N}{1-GFB}$$  \hspace{1cm} (22)

where $N$ is the length of a GV model subsequence; in this
study, it is equal to three. GFB is intended here as a form
of GFB pertaining to particular development methodologies (see Section 4-C for details). It can be
estimated using the mean biases of predictions of historical stage-effort sequences.

To obtain GFB, we need a set of historical projects that
comply with certain development methodologies. Each
project has a stage-effort sequence and is regarded as a
live project. First, we use (16) to make predictions stage
by stage for each project, then compute the bias of each
prediction. This process iterates until all projects are
processed. Finally, we average all the biases and assign
the result to GFB. The procedure for obtaining GFB is summarized in Fig. 10.

![Fig. 9. Model and prediction procedure.]

**Fig. 10. GFB obtaining function.**

**TABLE I. TYPES AND TREATMENTS OF STAGE-EFFORT SUBSEQUENCES**

<table>
<thead>
<tr>
<th>Class of original subsequence</th>
<th>Suitable model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Saturated</td>
<td>Verhulst</td>
</tr>
<tr>
<td>2 Exponential</td>
<td>GM(1,1)</td>
</tr>
<tr>
<td>3 Saturated after up flipping</td>
<td>Verhulst</td>
</tr>
<tr>
<td>4 Exponential after up flipping</td>
<td>GM(1,1)</td>
</tr>
</tbody>
</table>

**Function:** GetGFB, obtain the GFB (grey feedback) from a set of software stage-effort sequences.

**Input:** a set of stage-effort sequences.

**Output:** GFB

1) $j = 1$;
2) for each stage-effort sequence $\{x(1), x(2), ..., x(n)\}$, $n > 3$
3) for each $i$ from 3 to $n$:
4) $\delta(x + 1) = $ Gres. Predict $x(i - 1)$, $x(i)$;
5) $bias(j) = \frac{x(i + 1) - \delta(x + 1)}{x(i + 1)}$
6) $j = j + 1$;
7) end for
8) end for
9) $GFB = \frac{1}{j-1} \sum_{k=1}^{j} bias(k)$

**Fig. 10. GFB obtaining function.**

**F. Example**

We now demonstrate the complete prediction procedure
for a “live” project using the GV method. The initial
Step 1: Construct the model subsequence using the three most recent stage-effort values. The subsequence is \( X^{(0)} = (x^{(0)}(1), x^{(0)}(2), x^{(0)}(3)) = (260, 101, 450) \).

Step 2: Smooth the subsequence. Use (21) to transform the subsequence into: \( X^{(1)} = (x^{(1)}(1), x^{(1)}(2), x^{(1)}(3)) = (220, 448, 811) \).

Step 3: Determine the subsequence type. According to Section 5-C, because \( x^{(1)}(1) < x^{(1)}(3) \) and \( x^{(1)}(2) < (x^{(1)}(1) + x^{(1)}(3))/2 \), so \( X^{(1)} \) is an up-concave sequence, belongs to type 2, and according to Table I, the suitable prediction model is GM(1,1).

Step 4: Build the prediction model. According to the building procedure of GM(1,1) (see Section 3-B for details), first apply AGO on \( X^{(1)} \) and obtain the AGO sequence: \( X^{(1)} = (x^{(1)}(1), x^{(1)}(2), x^{(1)}(3)) = (220, 448, 811) \).

Then, we can get \( Z^{(1)} \), \( B \), and \( Y_N \) as:

\[
Z^{(1)}(1) = \begin{bmatrix} x^{(1)}(1)(2) \\ x^{(1)}(1)(3) \end{bmatrix} = \begin{bmatrix} \frac{x^{(1)}(2) + x^{(1)}(1)}{2} \\ \frac{x^{(1)}(3) + x^{(1)}(2)}{2} \end{bmatrix} = \begin{bmatrix} 334 \\ 620.5 \end{bmatrix}
\]

\[
Y_N = \begin{bmatrix} x^{(0)}(2) \\ x^{(0)}(3) \end{bmatrix} = \begin{bmatrix} 228 \\ 363 \end{bmatrix}
\]

\[
B = \begin{bmatrix} -x^{(1)}(2) & 1 \\ -x^{(1)}(3) & 1 \end{bmatrix} = \begin{bmatrix} -334 & 1 \\ -205.5 & 1 \end{bmatrix}
\]


\[
\frac{a}{b} = (B^T B)^{-1} B^T Y_N = \begin{bmatrix} 0.4569 \\ 75.4112 \end{bmatrix}
\]

Therefore, we can get the prediction model of \( X^{(1)} \) by (8):

\[
\hat{x}^{(1)}(k+1) = (\hat{x}^{(1)}(1), \hat{x}^{(1)}(2), \hat{x}^{(1)}(3)) = (443, 795, 1351)
\]

Note that \( \hat{x}^{(1)}(4) \) corresponds to stage 4 (the stage for which we are predicting).

Step 5: Make the preliminary prediction. According to (9), we can get the preliminary prediction of stage 4:

\[
\hat{x}^{(0)}(4) = \hat{x}^{(1)}(4) - \hat{x}^{(1)}(3)) = 1351 - 795 = 556.
\]

Step 6: Apply SAC to the preliminary prediction to obtain the optimum prediction. The final prediction value is:

\[556 \times \text{SAC} = 556 \times 0.76 = 422.\]
B. Experimental Methods

1) General Method: The purpose of the experiments is to evaluate the prediction performance of GV. First, we preprocess the four datasets (see Section 6-B2 for details), then for each of the four datasets, we systematically extract five pairs of the training and test datasets, and obtain a total of 20-pair training–test datasets. We obtain GFB and SAC on each training set, and evaluate GV on the test set (see subsection Section 6-B3 for details). Finally, we use a LR method and the Kalman filter (KF) method (see Section 6-B4 for details) as benchmarks to compare against the performance of GV.

2) Data Preprocessing: Inspection of the datasets reveals that there is a lot of noisy and inconsistent data. We preprocess the data as follows:

Delete the extreme cost data values from the stage-effort sequences. These values are too small or too big when compared with the other effort values. The stages with such extreme values are abnormal stages and could not be predicted using a generally useful model. For instance, the minimum value of stage effort is zero, and this is considered an extreme value (see Table III).

3) Validation Method: Cross-validation is a method for estimating generalization error based on “resampling” [33]. We use a five-fold cross-validation strategy as the validation approach. In five-fold cross-validation, the dataset \( D \) is randomly partitioned into five mutually exclusive subsets, \( D_1, D_2, \ldots, D_5 \), each of approximately equal size. The inducer is trained and tested five times. Each time \( t \in \{1, 2, \ldots, 5\} \), the subset \( D_t \) is reserved as the test set, and the remaining subsets \( D \setminus D_t \) are used as training set.

We obtain SAC from the training set (see Section 5-E for details) and evaluate GV on the test set. When evaluating, we regard the projects in the test set as live projects, i.e., for each project with stage-effort sequence \( \{x(1), x(2), \ldots, x(n)\} \), \( n \geq 3 \), we use \( \{x(k-2), x(k-1), x(k)\} \), where \( 3 \leq k \leq n-1 \), as the input subsequence, and use (18) to obtain the prediction value \( \hat{\epsilon}_{opt}^{(n+1)} \). For a project with \( n \) stages, we can make \( n-3 \) predictions. For each prediction, we compare the prediction value with the actual value to obtain evaluation results. In this study, we use the bias, mean magnitude of relative error (MMRE), and median MRE (MdMRE) as evaluation measures.

The Bias establishes whether models are biased and tend to over or under prediction. The Bias is defined as follows:

\[
Bias_i = \frac{\epsilon_i - \hat{\epsilon}_i}{\epsilon_i}
\]

where \( \hat{\epsilon}_i \) is the prediction value of actual effort \( \epsilon \).

The magnitude of relative error (MRE) is another common criterion for evaluating software effort prediction methods. For a prediction \( i \), the corresponding MRE is defined as follows:

\[
MRE_i = |Bias_i|
\]

By averaging MRE over multiple predictions \( n \), MMRE is obtained as

\[
MMRE = \frac{1}{n} \sum_{i=1}^{n} MRE_i
\]

MMRE is the most frequently used criterion for evaluating software effort prediction methods. However, it is known to be sensitive to individual predictions with excessively large MREs. We, therefore, also use MdMRE for the \( n \) predictions, which is less sensitive to extreme values as another measure. For both MMRE and MdMRE, a higher value means lower prediction accuracy.

4) Benchmark Methods: MacDonell and Shepperd employed a simple LR method in [12], and because there is no other work that can be compared, we use LR as a benchmark.

However, as LR often suffers from a leverage effect of outliers [17], we complement it with a further approach. The KF method [34] is a well-known series prediction method that can deal with noise and outliers robustly [35], so we also use the KF as another benchmark method and determine its arguments from the sequences in the training set (the training and test sets for KF used are same as those used with GV, see Section 6-B3 for details). For fairness, we use the same datasets and subsequences for the three approaches.

As many statistical techniques that deal with the prediction of time series data require large samples (i.e., long sequences), so they are not suitable to be benchmarks here. For example, effective fitting of Box–Jenkins models (often called autoregressive integrated moving average (ARIMA) models) [36] typically requires at least 50 observations [37]. This is too many for most software project stage-effort sequences and cannot be satisfied here.

C. Experimental Results

We conduct the experiments using GV (the GV
method), KF (the KF method), and LR (the simple LR method) on the four datasets. Tables IV–VI report the accuracy of the respective methods in terms of MMRE, MdMRE, Bias, and the improvements of GV upon KF and LR.

From Tables IV–VI, we observe that the MMREs, MdMREs, and Biases of GV are superior to those of KF and LR for all four datasets. Further, GV’s prediction accuracies across all datasets exhibit smaller differences. This indicates that GV has better consistency and stability on different development methodologies. The small bias values also imply that GV can cope with the differences in development methodologies to obtain optimal results. We also observe that compared with KF and LR, GV’s MMREs are lower by at least 28% and 50%, respectively, MdMREs are lower by at least 22% and 53%, respectively, and GV shows very good performance with respect to Bias, with values that are lower by at least 237% and 259% than KF and LR, respectively.

In statistics, percentiles are used to describe characteristics of distributions. Here, we used percentiles to explore the distribution of the absolute residuals of prediction accuracy for the three methods with four datasets. Table VII contains the results. From it, we can find that GV has less values than KF and LR for the 5th, 10th, 25th, 50th, 75th, 90th, and 95th percentiles with all four datasets except for the 75th percentile of dataset 4, LR has a less value. This reveals that GV outperformed both KF and LR.

Although we have made some observations based on the data presented in the tables before, to rigorously compare the differences between prediction accuracy between the three methods, we need to perform statistical significance testing. The MMREs of the three methods do not follow a normal distribution, so we use one-tailed Wilcoxon matched-pairs signed ranks tests to examine if there exist significant improvements on the MMREs of GV over the other two methods. Table VIII gives the testing results.

From Table VIII, we observe that all p-values are less than 0.001. This means that the MMREs of GV are significantly lower (and, therefore, better) than those of KF and LR.

To summarize, in this study, GV outperforms KF and LR on all four datasets and demonstrates considerable potential. The reasons lie in the following.

1) The AGO procedures of grey models make the original stage-effort sequences more regular, and thus, easier to fit.

2) The subsequence type recognition procedures makes GV flexible, which means GV can capture the various changing trends and further take full advantage of two grey models.

3) The capability of using GFB from historical projects means GV can utilize global domain knowledge to pilot local predictions along correct directions, avoiding reliance on just small local datasets and maximizing the capability of the method.

### Table IV. MMRE of GV, LR, and KF, and GV’s Improvement Upon KF and LR with Different Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MMRE (%)</th>
<th>GV’s MMRE Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>upon KF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upon LR</td>
</tr>
<tr>
<td>1</td>
<td>62.60</td>
<td>80.35</td>
</tr>
<tr>
<td>2</td>
<td>61.44</td>
<td>84.98</td>
</tr>
<tr>
<td>3</td>
<td>61.52</td>
<td>95.69</td>
</tr>
<tr>
<td>4</td>
<td>54.48</td>
<td>77.95</td>
</tr>
</tbody>
</table>

### Table V. MdMRE of GV, LR, and KF, and GV’s Improvement Upon KF and LR with Different Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MdMRE (%)</th>
<th>GV’s MdMRE Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>upon KF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upon LR</td>
</tr>
<tr>
<td>1</td>
<td>46.96</td>
<td>59.02</td>
</tr>
<tr>
<td>2</td>
<td>44.17</td>
<td>53.89</td>
</tr>
<tr>
<td>3</td>
<td>49.81</td>
<td>67.59</td>
</tr>
<tr>
<td>4</td>
<td>42.04</td>
<td>57.01</td>
</tr>
</tbody>
</table>

### Table VI. Bias of GV, LR, and KF, and GV’s Improvement Upon KF and LR with Different Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Bias (%)</th>
<th>GV’s Bias Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>upon KF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upon LR</td>
</tr>
<tr>
<td>1</td>
<td>-0.76</td>
<td>-25.73</td>
</tr>
<tr>
<td>2</td>
<td>0.18</td>
<td>-35.83</td>
</tr>
<tr>
<td>3</td>
<td>3.22</td>
<td>-40.42</td>
</tr>
<tr>
<td>4</td>
<td>5.85</td>
<td>-19.73</td>
</tr>
</tbody>
</table>

### 7. Conclusion

Dynamic software project stage-effort prediction facilitates the evaluation of potential effort problems, and could provide early warning information, thus ensuring that projects are completed within (possibly adjusted) schedules and budgets. In this paper, we have proposed a novel approach of using grey models of grey system theory to address the software stage-effort prediction problem during the development process. The proposed method can predict the future stage effort using the effort of three most recent continuous stages and can suit particular development methodologies by using a novel GFB mechanism.
Our experiments have been conducted on a large-scale software engineering data repository split into four datasets based on development methodology. Because there is no other work that can be compared, we employed the KF method and the LR method as benchmarks. The results show that the proposed GV method outperforms KF and LR in terms of MMRE, MdMRE, and Bias for all datasets used.

Finally, we pose the question of generalization. The datasets are drawn from a repository of medium to large projects from an international software house. They cover a range of application areas such as commerce, government information systems, defense, and retail. Of course, different organizations may run projects and collect data in different ways. Therefore, it would be interesting to see replication of this study in different environments. Nevertheless, this is an encouraging result and shows that the method has considerable potential.

## REFERENCES


AUTHORS

Yong Wang received the M.S. degree in computer science in 2003 from Xi’an Jiaotong University, Xi’an, China, where he is currently working toward the Ph.D. degree in computer science. He was with a Research Institute in Qingdao, China, and has more than ten years of experience in software development and project management. His current research interests include software cost estimation, software project management, and software engineering data quality.

Qinbao Song received the Ph.D. degree in computer science from Xi’an Jiaotong University, Xi’an, China, in 2001. He is currently a Professor of software technology in the Department of Computer Science and Technology, Xi’an Jiaotong University, where he is also the Deputy Director of the Department of Computer Science and Technology. He is also with the State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China. He has authored or co-authored more than 70 referred papers in the area of machine learning and software engineering. He is a board member of the Open Software Engineering Journal. His current research interests include machine learning, empirical software engineering, and trustworthy software.
Stephen MacDonell received the Ph.D. degree in software engineering from the University of Cambridge, Cambridge, U.K., in 1993. He is currently a Professor of software engineering and the Director of the Software Engineering Research Laboratory (SERL) at Auckland University of Technology (AUT), Auckland, New Zealand. He is currently engaged in research in software metrics and measurement, project planning, estimation and management, software forensics, and the application of empirical analysis methods to software engineering datasets. Prof. MacDonell is a member of the IEEE Computer Society, the Association for Computing Machinery (ACM), and the Editorial Board of Information and Software Technology.

Martin Shepperd received the Ph.D. degree in computer science from the Open University Milton Keynes, U.K., in 1991. For a number of years, he was with a bank as a Software Developer. He is currently a Professor of software technology at Brunel University, London, U.K., where he is also the Director of the Brunel Software Engineering Research Centre (B-SERC). He has authored or co-authored more than 100 refereed papers and three books in the areas of software engineering and machine learning. From 1992 to 2007, he was the Editor-in-Chief of the Journal of Information and Software Technology. He is currently an Associate Editor of the Journal of Empirical Software Engineering. Prof. Shepperd was an Associate Editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (2000–2004).

Junyi Shen graduated from the Department of Applied Mathematics, Xi’an Jiaotong University, Xi’an, China, in 1962. He is a Professor of software technology in the Department of Computer Science and Technology, Xi’an Jiaotong University. He has authored or co-authored more than 100 refereed papers and three books in the areas of database, data mining, and software engineering.