Semi-Quantitative Modeling for Managing Software Development Processes

He Zhang[1,2], Jacky Keung[1,2], Barbara Kitchenham[3], Ross Jeffery[1,2]
[1]*School of Computer Science and Engineering, University of New South Wales*
[2]*National ICT Australia*
[3]*School of Computing and Mathematics, Keele University*
[1,2]{he.zhang, jacky.keung, ross.jeffery}@nicta.com.au, [3]b.a.kitchenham@cs.keele.ac.uk

Abstract

Software process modeling has become an essential technique for managing software development processes. However, purely quantitative process modeling requires a detailed understanding and accurate measurement of software process, which relies on reliable and precise history data. This paper presents a semi-quantitative process modeling approach to model and manage software development processes. It allows for the existence of uncertainty and contingency during software development, and facilitates a manager's qualitative and quantitative estimates and assessments of process progress. We demonstrate its value and flexibility by developing semi-quantitative models of the test-and-fix process of incremental software development. Results conclude that the semi-quantitative process modeling approach can support process or project management activities, including estimating, planning, tracking and decision making throughout the software development cycle.

1. Introduction

Software process modeling and analysis has been identified as being a vehicle for controlling development costs, duration, and achieving product quality. It has been widely accepted that improving the software process will also improve the software product quality, and increase project success rate. In recent years, software process modeling has been a major focus of software process improvement and is becoming one of the essential techniques for effectively managing software development processes. Nevertheless, purely quantitative process modeling requires very detailed understanding and accurate measurement of the software process, which relies on reliable and precise history data. Furthermore, in most cases these data are not readily available.

The semi-quantitative view, which is based on specifying quantitative intervals rather than single values, is not completely new in software engineering practice. For example, DeMarco reported that software projects are sometimes considered successful when the overruns are held to within thirty percent [1], and McConnell used a similar argument but suggested 25% [2]. However, the first formal introduction of semi-quantitative modeling and simulation into software engineering was done in our previous work using the classic software staffing problem [3]. This paper provides further evidence of the value of semi-quantitative modeling by specifying a different process and discussing how it could be used in practice. In this paper, we demonstrate how an incremental development process model is developed and refined by using the qualitative and semi-quantitative modeling paradigms, and explain how to apply this approach to manage software processes.

In Section 2, we present a brief discussion of incremental development and introduce semi-quantitative modelling. In Section 3, we develop qualitative models of the incremental development process focusing on the test-and-fix process and show how to introduce quantitative constraints to the qualitative models. Section 4 discusses how semi-quantitative process models support process management. Section 5 presents a practical example illustrating the outputs of the semi-quantitative models. We discuss our results in Section 6 and present our conclusions in Section 7.

2. Background and approach

2.1. Software process modeling and simulation

Osterweil identified two complementary types of software process research [4], which can be characterized as macro-process research, focused on

1530-0803/08 $25.00 © 2008 IEEE
DOI 10.1109/ASWEC.2008.50

phenomenological observations of external behaviors of processes, and micro-process research, focused on the study of the internal details and workings of processes. At the macro-process level, a process model is composed of a set of equations whose variables describe a process. The input variables deal with the observable and manageable aspects of the process.

In this paper, we model software processes semi-quantitatively with identification of the inherent cause-effects in the process. At the current time, our efforts concentrate on the macro-process. The following sections go through the semi-quantitative modeling approach for an incremental development process, and show how this model is developed and can be used for managing the software development activities.

2.2. Incremental development processes

Incremental development is a broad term describing a type of software development process, which includes iterative development, versioned implementation, spiral model, etc [5]. The basic idea is to divide the development into several smaller increments, which are gradually accumulated to become the complete system.

Basically, there are three phases in one increment. They are analysis, implementation, and testing. In this article, we only model the implementation and test-and-fix processes of an intermediate increment. After the current release is implemented, the progress enters a test-and-fix process, where this release is thoroughly tested and corrected. During this period, no new functionality is added into the release. The purpose of the process is to make sure that each release provides a robust foundation for its succeeding releases.

2.3. Qualitative & semi-quantitative modeling

Semi-quantitative modeling is implemented in two stages: first the system is model qualitatively, and then the quantitative constraints are placed on the model. Figure 1 shows how a system described by single QDE (qualitative differential equation) enables semi-quantitative modeling and simulation.

Qualitative models reflect the systems in the real world at an abstract level. Fewer assumptions are required than for quantitative models. However, all the assumptions must be first specified textually prior to converting them into a graphical notation, and finally into a formal specification in the modelling language. This allows the model to be validated at the textual level and for each subsequent representation of the model to be verified against the preceding representation. Qualitative models can be used for process simulation by the QSIM [6]. The output generated is a set of possible qualitative behaviors and each behavior consists of a sequence of states that describe open temporal intervals or time points [7]. These states present the system behavior from its initial state to final point. Time is treated as a qualitative variable in the model. The qualitative landmarks are created when necessary, they indicate critical points of the model parameters.

Quantitative constraints use bounding intervals to represent partial quantitative knowledge. Q2 is the semi-quantitative extension to QSIM. Given quantitative interval bounds on the values of some landmarks and envelope functions, Q2 defines a constraint-satisfaction problem (CSP). A solution to CSP is an assignment of a value range to each landmark consistent with the constraints. This reduces the number of valid behaviors since a contradiction refutes any inconsistent qualitative behavior and all its associated behaviors [7].

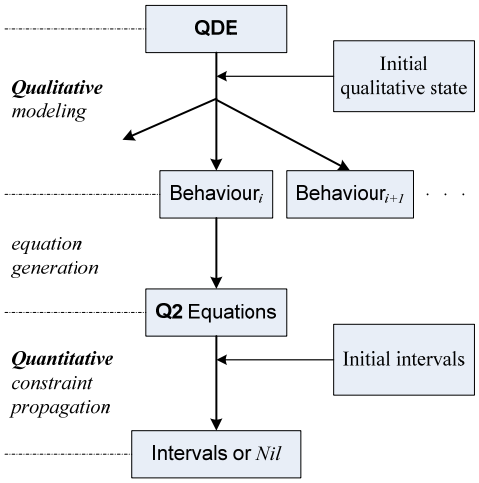


Figure 1. Semi-quantitative modeling and reasoning

3. Semi-quantitative models for test-and-fix process of incremental development

In the context of incremental development, the primary objective of the test-and-fix process model is to examine whether the increment can be completed within the desired time frame with the required quality. Other objectives are also addressed in Section 4. This section presents qualitative models for the implementation of an intermediate increment and its associated test-and-fix process. We show how quantitative constraints can be added to the qualitative models to obtain semi-quantitative models. In the

following sections, we use '*error*' as the equivalent term to '*defect*'.

3.1. Related models

The software testing process using quantitative modeling and simulation has been investigated by researchers. This section provides brief descriptions and comments of these models separately.

1). Abdel-Hamid and Madnick (AHM) modeled the basic software testing process, which is a sector of their integrated software process model with System Dynamics [8]. There are a few limitations in their model. For instance, their model is based on the waterfall testing process, rather than incremental testing process. In addition, there is only one explicit type of error, namely "Error" in the model. Furthermore, they did not isolate the newly generated errors and the residual errors from the test-and-fix process of the previous increments, which influence the current testing performance. Finally, their model neglects any switchover phenomenon of defect fixing productivity.

2). Huff *et al.* developed an alternative causal model for the test-and-fix process of incremental development [9]. They use quantitative equations to quantify the relations. Unfortunately, their models did not identify the impact of remaining active errors in the succeeding increments.

3). Tvedt developed a comprehensive process model of concurrent incremental development [10]. He considered the impacts on defect creation from engineer's capability, technical risk, and inter-dependency among the concurrent increments. However, active and passive errors were not explicitly handled in his model.

3.2. Qualitative modeling

The qualitative models consist of two interlinked models, which are developed to model the implementation (error generation) process and test-and-fix (error detection and correction) process for producing an intermediate release.

At the qualitative modeling stage, we investigate the incremental development processes, and abstract error-related features (assumptions) in qualitative form from them.

3.2.1. Modeling implementation process. One widely used linear model for software implementation is employed as the basic skeleton of this model, i.e. given workforce (*WF*), release size (*S*), and unit productivity (*PD*), the elapsed calendar days to complete the release can be calculated by $S/(WF*PD)$. However, because we are interested in the processes related to error generation and detection during each increment, more features related to error generation need to be involved in this model. In addition, the management overheads (including communications, adaptations, staff absences, configuration management, etc.) need to be considered as well when calculating the elapsed time.

During the implementation phase, there are two basic types of errors generated: active errors and passive errors. "Active errors" are errors which unless detected and retired will generate additional errors (active and passive) in subsequent increments. "Passive errors" remain in code until they are detected and retired but do not give rise to additional errors in subsequent increments.

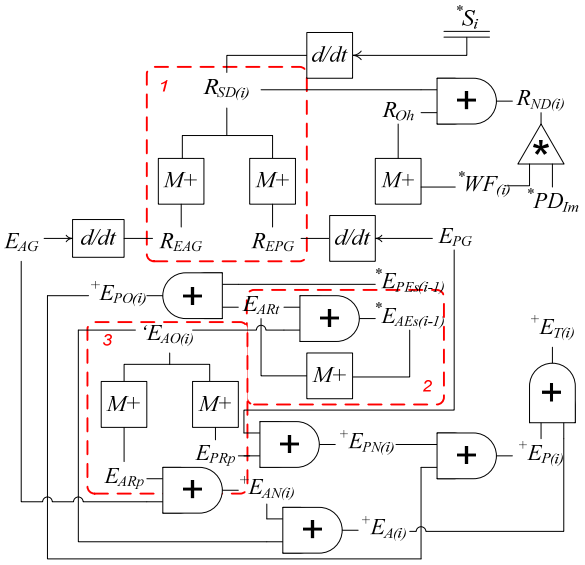
We include active and passive errors in the incremental implementation process because they impact the number of errors introduced into the incremental test-and-fix process. If we suppose the system is developed with an incremental top-down strategy, then in the early releases, most of errors committed are in the core or high level components and become active. If these errors are not detected, they tend to propagate through the succeeding increments that build on one another.

Therefore, the errors generated through each increment arise in two ways. The first is through the development of the construction of new functionality required in each increment (increment size, S_i); the second is the errors generated by existing active errors. However, for many undetected active errors, the reproduction will not continue after producing one or two "generations" of errors [8]. In this case, they effectively become undetected passive errors.

The qualitative assumptions for modeling the implementation process are explicitly given below:

1. All resources are focused on one increment, i.e. increments are sequentially linked with each other, and there are no concurrent increments at any time.
2. The size of current release (S_i) does not change during current increment.
3. Current release size (S_i) includes the necessary work, e.g. design, coding, unit testing and rework except system testing and defect fixing in this release.
4. More old active errors (E_{AO}) from previous releases will reproduce more active and passive errors (E_{ARp} and E_{PRp}) in current implementation.
5. Increasing software development rate (R_{SD}) increases both active and passive error generation rate (R_{EAG} and R_{EPG}).

6. Increasing the team size (WF) leads to a larger implementation overheads (R_{Oh}).
7. The average development productivity (PD_{Im}) does not change during current implementation.
8. A fraction of remaining active errors (E_{ARt}) are retired to become old passive errors (E_{PO}) in each intermediate release.



- $S_{(i)}$: Size of the ith increment
- R_{SS}: Software implementation rate
- R_{ND}: Nominal development rate
- R_{Oh}: Management overheads
- $WF_{(i)}$: Workforce for the ith increment
- ED_{As}: Active error density
- ED_{Ps}: Passive error density
- PD_{Im}: Implementation productivity
- R_{EA}: Active error generating rate
- R_{EP}: Passive error generating rate
- E_{AG}: Active error number generated
- E_{PG}: Passive error number generated
- $E_{AEs(i)}$: Escaping active errors from increment i
- $E_{PEs(i)}$: Escaping passive errors from increment i
- E_{ARt}: Retired active errors
- $E_{AO(i)}$: Old active errors in the ith increment
- $E_{PO(i)}$: Old passive errors in the ith increment
- $E_{AN(i)}$: New active errors in the ith increment
- $E_{PN(i)}$: New passive errors in the ith increment
- E_{ARp}: Reproduced active errors
- E_{PRp}: Reproduced passive errors
- $E_{A(i)}$: Active errors in the ith increment
- $E_{P(i)}$: Passive errors in the ith increment
- $E_{T(i)}$: Total errors in the ith increment

Figure 2. Qualitative implementation process model

Based on the above qualitative assumptions, the qualitative model is given in Figure 2, where the asterisk denotes an input parameter, and the plus sign indicates an output parameter.

3.2.2. Modeling test-and-fix process. The objective of test-and-fix process is to achieve an "acceptable level of quality", meaning that a certain percentage of errors will remain unidentified at release of the software [11]. During the incremental development, a small percentage of errors may escape from the current test-and-fix process, and remain in the implementation of the next increment.

In the test-and-fix process, a specific test suite is run and analyzed, detected defects are reported, and eventually fixed. In practice, the test cases are usually performed by a standalone team to avoid any potential bias. The work of correcting defects mostly falls to the team who implement the design and coding.

The test suite contains multiple test cases, which are prepared in advance, and ready prior to test-and-fix. During the test-and-fix, we assume that the black-box testing strategy is applied, since we assume that unit testing (based on white box testing) takes place during the implementation process. The time spent on detecting errors depends on the size of test suite (or the number of test cases in queue) for the current release, instead of the release size (lines of code or function points). However, the time spent on correcting defects relates to the number of detected defects.

Huff *et al.* argued that the defect fixing rate (productivity) slows as average length of defect queue drops below a certain number [9]. This switchover phenomenon normally happens near the end of defect fixing work, when a small number of defects are in the queue for developers. We use a single multiplier to reflect the productivity drop.

The qualitative assumptions for modeling the test-and-fix process are summarized as:

1. Test suite (Ts) is prepared before test-and-fix starts, and does not change during the process.
2. Errors (active and passive, old and new) are uniformly distributed across the test suite for the current release. In the other words, given the implementation and test suite, completing more test cases produces more detected defects.
3. Errors generated in the current release (E_{AN} and E_{PN}) have a higher probability of being detected with the same test case than the old errors (E_{AO} and E_{PO}) remaining from previous releases.
4. The average effort required to correct defects (PD_{Fx}) is constant throughout the process.
5. The team clearing defects is the same team developing the current release.
6. Increasing the team size leads to larger test-and-fix overheads (R_{Oh}).
7. All detected defects (E_{Fx}) in the current release must be cleared before the next release.
8. There is no new error generated (badfix) during defect fixing.
9. The defect fixing rate (R_{NF}) is a multiple of the productivity rate (PD_{Fx}), the work force size (WF) and a multiplier (m_f) which decreases after the switchover.

Here we focus on the "new" and "old" errors in the test-and-fix process model (see Figure 3), as they are associated with different possibility (defect hitting rates) and effort to detect them, which ultimately influences the performance of the testing process.

The model can have multiple exits when performing simulation. This process is not completed until all detected defects are fixed, or a required percentage of test cases are passed within a desired period, and so on.

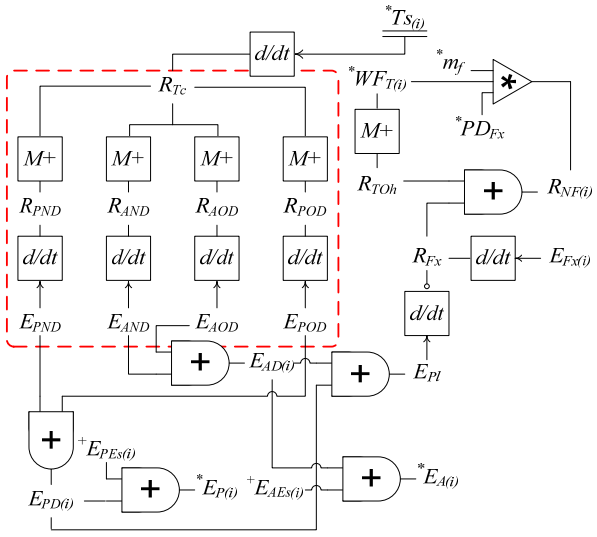


Figure 3. Qualitative test-and-fix process model

These two models connect each other iteratively to model the entire incremental development. Four QDEs were coded to represent the above qualitative models: QDE A for implementation process model, and QDE B, C, & D for test-and-fix process model. Figure 4 shows the connections among these QDEs. This formal representation is required for the further execution of the qualitative model.

Transition 1 (tp_1), when the implementation phase finishes; *Transition 2* (tp_2), when the accumulated detected defects reach the threshold that triggers bug-fixing; *Transition 3* (tp_3), when the average length of defect queue for developers drops to switchover point; *Transition 4* (tp_4), when detected defects are fixed, and current release done.

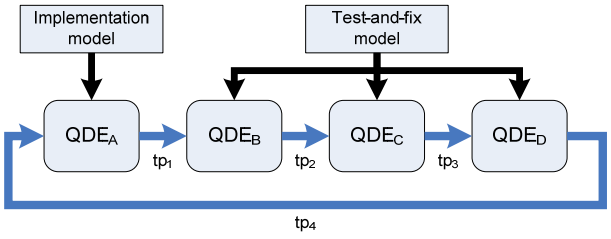


Figure 4. Transitions between QDEs

3.3 Quantitative Constraints

When moving to a semi-quantitative model, the qualitative model is quantified by the partially quantitative equations (Q2). Q2, including intervals and envelope functions, are the representations of incomplete quantitative knowledge and uncertain scenarios.

Envelope functions. Monotonic functions ($M+$ and $M-$ constraints) may be restricted to take on values within an envelope defined by upper and lower edges. The edges are each defined by a quantitative function and its inverse function [7]. The nominated functions can be linear or nonlinear. A pair of qualitative values satisfies the quantitative constraints if the intervals associated with its landmarks fall within the envelope.

Parameter intervals. Unit and unit conversion were not addressed in the qualitative model because the landmark values are symbolic names for unknown real values (like algebraic variables). All quantities are assumed to have appropriate and compatible units. However, because some quantitative interval bounds are involved in quantitative constraint, the units must be explicitly denoted for the interval arithmetic. The interval span mainly depends on the completeness, consistency, and certainty of our knowledge or empirical data. It could be narrow for a software process expert in a mature organization, or conversely, broad for the novices in an ad hoc project.

3.3.1. Quantifying implementation process. As indicated in Figure 2, there are six monotonic functions in the qualitative implementation model. Thus, we need to specify the envelope functions for them separately.

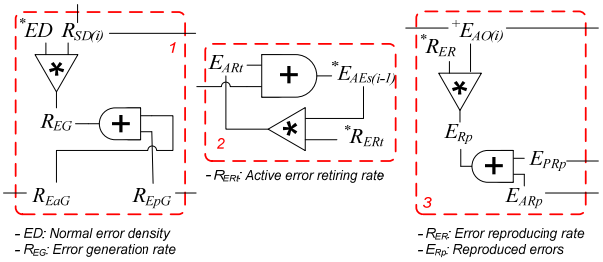


Figure 5. Refinement of portions of implementation model

To simplify the discussion based on the above-mentioned assumptions, we assume linear relationships between software development rate and error generation rate, between residual active errors and retired errors, and between old active errors and propagated errors (indicated by dashed line in Figure 2). Therefore, some auxiliary parameters need to be introduced with the quantitative constraints: error densities (ED) in Portion 1, error retirement ratio (R_{ERt}) in Portion 2, and error reproducing rates (R_{ER}) in

Portion 3 (Figure 5). The discussion of the relation between team size (*WF*) and overheads (R_{Oh}) can be found in [12] and [3].

Based on the above assumptions, the envelope functions for these monotonic constraints can be converted to the assignment of value ranges. For a ten-increment project, with reference to [8], active error retirement ratio can be quantified by the value ranges in Table 1.

Table 1. Value ranges for active error retirement rate

Incrment	1,2,3	4,5,6	7,8	9	10
R_{ERt}	[0, 0]	[.05, .15]	[.2, .4]	[.5, .9]	[1, 1]

3.3.2. Quantifying test-and-fix process. In the qualitative test-and-fix model (Figure 3), four types of errors, the combinations of new-old and active-passive error types, are modeled separately. The new errors and old errors are associated with different hitting rates (Assumption 3, Section 3.2.2) in testing. Nevertheless, the qualitative model provides no indication of this difference, which needs to be quantified here.

The error detection rate may change across the test cases. Its value depends highly on the applied testing strategy, the design of the test case, and the portfolio of the test suite. In the context of quantitative constraints, we postulate that the hitting rates do not change across the testing cases. Then we can use two hitting rates (new error hitting rate h_n and old error hitting rate h_o, $h_n > h_o$) and corresponding error detection rates to specify the relations (Figure 6). Their value ranges can be calculated based on the history records, such as [.85, 1] (h_n) and [0, .1] (h_o) of nominal detecting rate.

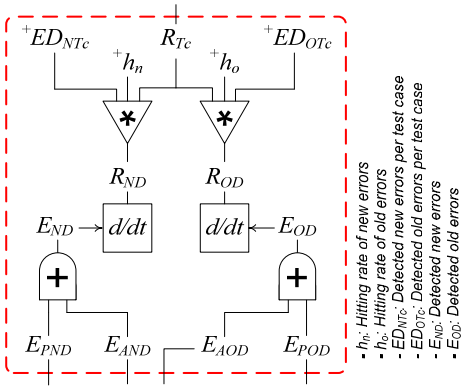


Figure 6. Refinement of portion of test-and-fix model

The estimate of multiplier (m_f) is based upon the time period when the average length of a developer's defect queue falls below one particular value. Huff *et al.* argued that typically a developer spends one quarter of time waiting for a defect to arrive, and three quarters handling a single defect in the queue [9]. However, this value is difficult to directly observe and measure. It is therefore more sensible to define the range of value (like [.7, .8]) for a general context.

4. Managing software process with semi-quantitative models

DeMarco identified two board categories of solutions for managing software projects: *political solutions for political problems*, and *technical solutions for technical problems* [1]. In this section, we only discuss semi-quantitative modeling in support of the "technical solutions".

4.1 Estimating

A process model is a system with inputs and outputs. We denote the input and output parameters in the qualitative model and quantitative constraints. The inputs can be further classified into direct and indirect types. The values of direct inputs can be quantitatively and explicitly specified or measured, which enables a manager to allocate, observe and change their values, such as T_S, WF_T, E_F, h_o, and h_n (see Figure 3 & 6).

On the other hand, some indirect inputs cannot be directly observed, measured, or quantified, such as R_{ERt}, R_{EAR}, PD_{Fx}, and m_f (see Figure 3 & 5). However, these are required to produce model outputs. Although some statistical techniques, for example control charts [13], to some degree provide for the estimation based on analysis of historical process data, blindly applying single-point value assignment (especially for the complicated model) may result in the dramatic uncertainty propagation, and unrealistic outputs.

In contrast, semi-quantitative modeling is capable of handling the estimation of uncertainty and imprecision on indirect input parameters (see Section 6).

4.2 Planning

Planning is an iterative estimating and refining process. One semi-quantitative solution for planning at the project level can be found in our previous work [14]. Using a process model, the planning process consists of assigning values to the input parameters of the model, and then computing or reasoning about the output parameters as predicted results for planning.

Semi-quantitative modeling can speeds up this iterative process by using interval arithmetic and constraint propagation techniques, rather single-point value calculation [7]. A factor-effect matrix (example as Table 2) needs to be developed with the qualitative process model [14]. For example, increasing test suite

size (T_S) may decrease the number of escaping defects (E_{AEs}, E_{PEs}), but also increase the process duration (t). The factor-effect matrix is used for fine tuning the numeric intervals to achieve the desired outputs between iterations quickly.

Table 2. Factor-effect matrix of test-and-fix model

input parameters \ output parameters	E_{PEs}	E_{AEs}	E_{AD}	E_{PD}	E_F	t
T_S	-	-	+	+	+	+
WF_T	N	N	N	N	N	-
PD_{Fx}	N	N	N	N	N	-
m_f	N	N	N	N	N	-
E_P	+	+	+	+	+	+
E_A	+	+	+	+	+	+
ED_{ANTc}…	N…	-…	+…	N…	+…	+…
h_o	-	-	+	+	+	+
h_n	-	-	+	+	+	+

+: positive relation -: negative relation N: no explicit relation

This allows a manager to learn from the matrix and the iterations how a combination of inputs affects the outputs, and how sensitive the model (process) is to particular parameter changes.

By considering multiple uncertain factors, semi-quantitative modeling does not force you accept some "accurate" single-point estimates during planning (actually you can turn value range to single-point by assigning 0 to interval). Instead, given the intervals of inputs, it produces the value range you can expect.

4.3 Tracking

The behavior tree is an important outputs generated by semi-quantitative modeling and reasoning. In contrast with traditional methods, it depicts the road map with all possible alternative routes of the process. Figure 7 shows a behavior tree for one scenario of the test-and-fix process. Each dot point indicates one critical time point during the process progress. The transition points (*tp*) are denoted by landmarks "Φ". For instance, the first *tp* of each behavior (branch) means that the defect correction starts when the length of defect queue reaches the threshold; the second indicates the switchover point; and the test-and-fix process ends at the third.

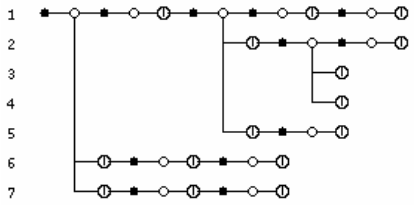


Figure 7. Behavior tree for one scenario of test-fix process

All critical time points and transition points need to be identified as the check-points for process tracking. The corresponding control metric table can be created for each measurable parameter based on the predictions of the behavior tree. The table can help a manager observe and judge if the process is under control.

4.4 Decision making

No decision making is needed if the process lives up to its prediction. However, progress often diverges from expectations. By using behavior trees and control tables, decisions can be made at check-points when the process is not on track.

Decision making answers to a series of "what if" questions, such as what if we increase the staffing level, or extend the time period? Semi-quantitative modeling can maintain the integrity of the prediction when a manager struggles to cope with multiple uncertain factors and has to make a tradeoff.

Although the process performance indicators are represented with value ranges in semi-quantitative modeling, it also helps to extract qualitative assessment from them for decision making and management.

5. Example

5.1 Baseline project

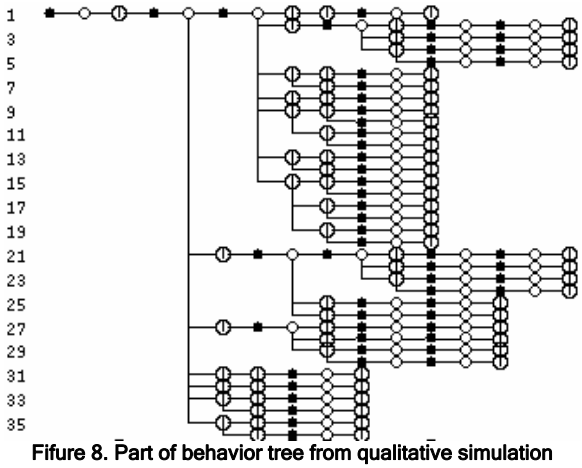
In order to assess the performance of the semi-quantitative model we need a baseline project for comparison. When using the qualitative simulation approach, a set of qualitative behaviors is generated, which are difficult to compare with one specific project. The baseline project for quantitative comparison is derived from data reported by Tvedt [10]. The characteristics of this project are given in Table 3. The requirements size of the baseline project was 80,000 lines of code written in COBOL. As the incremental development is applied in this project, it caused 10,667 lines of code as the estimated overheads, which results in the total project size of 90,667 lines of code. The project was scheduled to last one calendar year, with a workforce of 15 full-time engineers from start to finish.

Table 3. Characteristics of the baseline project

Attributes	Values
Project size	90,667Loc
Number of increments	3
Increment 1	26,667Loc
Increment 2	32,000Loc
Increment 3	32,000Loc
Project schedule	250days (1 year)
Project team size	15 experienced engineers
Estimated budget	3750 man-days

5.2 Qualitative simulation

Given the qualitative assumptions and QDE models, no specific numeric values are needed to perform qualitative simulation. We can simulate one intermediate increment by assuming the volume of escaped active and passive errors (E_{AES}, E_{PES}) from previous releases qualitatively. The simulation generates 72 possible qualitative behaviors for this intermediate increment. Figure 8 shows part of the behavior tree containing behavior 1 to 36. One example behavior (Behavior 26 of 72 in Figure 9) depicts the trends and states of four key variables in Figure 8. The difference between the first half of behaviors (Behavior 1 to 36) and the second half (Behavior 37 to 72) is whether all escaping active errors from previous release are partially retired or entirely retired. In terms of the difference of this qualitative initial condition, the behavior tree is branched at its root. The four transition points ("Φ") through each course indicate the landmarks of one increment process (see Figure 4).



Fifure 8. Part of behavior tree from qualitative simulation

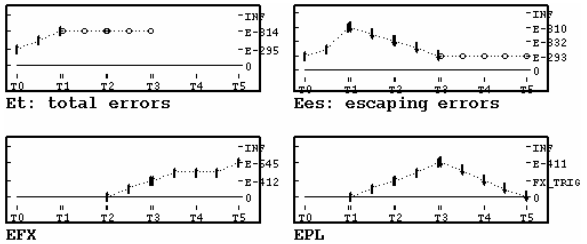


Figure 9. Behavior 26 of 72

Most of the qualitative behaviors observed by simulation represent the complicated relationships in the variable space, including E_{AEs} vs. E_{AN}, E_{PEs} vs. E_{PN}, E_{Es} (E_{AEs} + E_{PEs}) vs. T_S, and R_{FX} vs. R_{Tc}, etc. For example, the value of E_{AEs} can be increased due to a low active error retirement ratio R_{ERl} and a high active error generating rate (R_{EA}) or a high active error propagation rate (R_{EAR}), and vice versa.

5.3 Semi-quantitative simulation

Tvedt's model used the project shown in Table 3. The variables in test-fix process are also affected by auxiliary factors from other related sectors, such as the human resource. Our semi-quantitative model treats different types of defects separately, gains more insight into error propagation across increments, and allows some ignorance of the impacts from other sectors.

By applying the semi-quantitative test-and-fix process model on the baseline project, the simulation is able to generate one semi-quantitative behavior for each increment. The behaviors of some key variables are plotted in charts of Figure 10 (where Inc1 is denotes by *a*, Inc2 by *b*, and Inc3 by *c*).

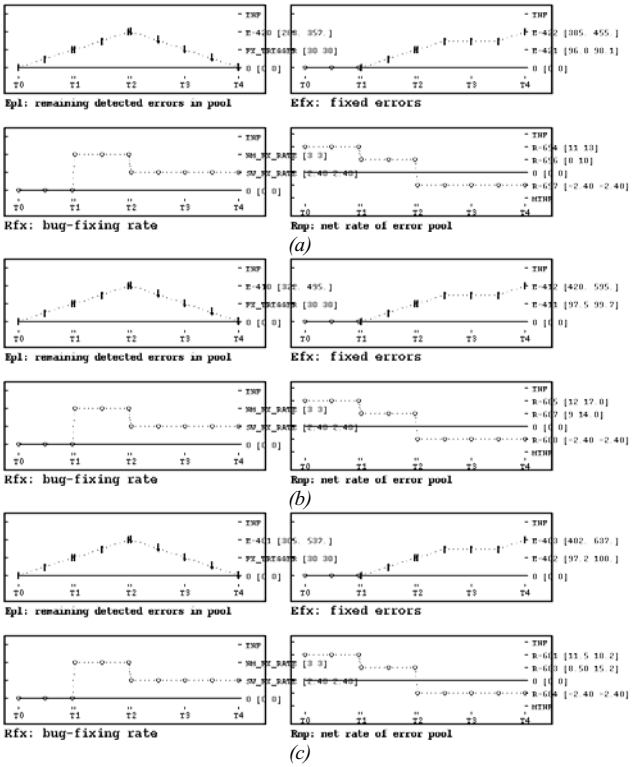


Figure 10. Semi-quantitative behavior of test-and-fix process

In contrast to our model, Tvedt's model simulated the concurrent development that allows overlapping between increments. Thus, it is not necessary to compare the elapsed times. We present only their quality features for the brief comparison in Table 4. Our model estimated that the project can produce the software product with no more than 414 defects, which

is consistent with the simulation result generated by using Tvedt's model.

Table 4. Simulation results and comparison

	E_N	E_{Fx}	E_{Es}	Density
Inc 1	[480, 533]	[385, 455]	[25, 148]	<5.5KLoc
Inc 2	[598, 781]	[420, 595]	[3, 361]	<6.1KLoc
Inc 3	[576, 816]	[402, 637]	**[0, 414]**	**<4.2KLoc**
Tvedt's model			**279**	**3.5/KLoc**

It is noticeable that the error value ranges extend significantly during Increment 2. This is mainly caused by the increased uncertainty of active error generation ratio (rR_{EAG} in Table 5). As there are only three increments in the baseline project, it leads to the difficulty in estimating the ratio between R_{EAG} and R_{EPG}, which might be simply assumed in purely quantitative modeling.

Table 5. Value ranges of rR_{EAG} for 3 increments

Increment	1	**2**	3
rR_{EAG}	[.95, 1.0]	**[.2, .9]**	[0, .25]

The quantitative constraints that were applied in our semi-quantitative model are mostly defined based on literature and experience data, which have yet been calibrated with the baseline project. In this example, we only use the baseline project information to specify the initial state (initial value ranges of input variables) and control the simulation (transitions and iterations). The simulation results demonstrate that this approach can produce reasonable prediction ranges when the specific process information is incomplete, uncertain.

6. Discussion

6.1 Feasibility considerations

Qualitative modeling. The modeling process (from qualitative assumptions to QDEs) offers a basic structure of the process and abstracts its pertinent characteristics. It serves as a vehicle to enhance the qualitative understanding of specific process, and further the model is able to accommodate the qualitative agreement among the stakeholders involved in development. The qualitative models can also be shared and reused across software organizations, no matter at what maturity level they are [15]. Therefore, qualitative modeling can be used to extract and manage common knowledge and insights of software processes at a high level.

Quantitative constraints. Because the qualitative model reflects the skeleton of the problem under investigation, in most cases, there is no need to amend the qualitative model. Instead, managers focus more on the quantitative constraints (with their gradually enriched experience) which improve process estimation and measurement.

Furthermore, quantitative constraints provide a means to continuously refine the qualitative model with improved knowledge. Different organizations, or even different projects, can develop their own specific quantitative constraints and extensions based on the same qualitative model.

6.2 Alternative approaches

There are two broad categories of methods dealing with uncertainty: probabilistic and non-probabilistic [16]. Semi-quantitative modeling falls into the latter.

Monte Carlo Method. Monte Carlo simulation is one popular quantitative method for analyzing software process. Although it takes many samples of the value range, unfortunately, they are still a finite set. Thus, this method cannot guarantee all possibilities fall into their solution. This problem turns to be more subtle when more factors change simultaneously, which may result in missing some important behaviors. Furthermore, the computational cost of using Monte Carlo method dramatically increases when dealing with a large variable space. In contrast, the cost of semi-quantitative simulation does not depend on the size of the variable space. It is a function of the number of distinct qualitative behaviors predicted [7]. Moreover, semi-quantitative modeling allows for the existence of many uncertain elements without needing to assume their statistical distribution over the range.

Statistical Process Control (SPC). This control technique, as described by Florac and Carleton [13], provides useful method and tool to improve the controllability of development process. Graphic tools, e.g. control charts and capability histograms, are used to analyze the process. The applicability of SPC is independent of the life cycle model and development methodology. However, it does require that a rigorous measurement process be instituted with the development process. In addition, when the number of parameters that affect the process is relatively high, the combinations of possible values are even higher and analysis of all alternatives becomes very difficult, if not impractical.

Fuzzy Logic. As another non-probabilistic method, Fuzzy logic provides a typical set-valued quantitative approach. It describes the real world system with fuzzy set, which is a fuzzy subset of the universe of discourse [16]. It applies a rough boundary to handle the uncertainty, and the mapping to fuzzy set is in an arbitrary way, linear or nonlinear. In contrast, semi-quantitative modeling describes the system boundary with real numeric values, which maintains the

precision while coping with uncertainty. Each approach possesses its advantages and limitations. The selection depends on user's capability and requirements.

Semi-quantitative modeling performs modeling and simulation by refinement: define a set of possible solution, and shrink it by cutting out the illogical behaviors. This approach guarantees integrity of the solution. In addition, this approach produces not only the final states of project, but all possible process behaviors (routes) consistent with the constraints of value ranges and envelope functions, which can be used for in-progress process tracking.

Semi-quantitative modeling still requires the process to be observed and measured quantitatively. Plus, it does not mean assigning a value rang to every parameter, and not an envelope to every equation. It provides a manager an alternative modeling approach and flexibility in managing process. The capability to apply finer intervals reflects the organization's maturity. In addition, semi-quantitative modeling can be used for retrieving qualitative assessment of the process performance for management.

7. Conclusions

This paper reports a study of modeling incremental development focusing on test-and-fix process, and proposes the process modeling by using semi-quantitative approach, which is demonstrated as a powerful supplementary technique to quantitative modeling. It provides a progressive refinement of process model with reference to incomplete quantitative knowledge. Some considerations on how this novel approach improves process management are also discussed.

The future efforts on this modeling approach can be carried out on the following aspects:

1. Extending this semi-quantitative process model by considering more factors, e.g. bad-fixing, test suite regression, and increment dependency.
2. Investigating the semi-quantitative process modeling at micro-process level with detailed definition of low-level processes.

8. Acknowledgements

National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council. Funds were also provided by the University of New South Wales.

References

[1] T. DeMarco, *Controlling Software Projects: Management, Measurement & Estimation*. New York: Yourdon Press, 1982.

[2] S. McConnell, *Software Estimation: Demystifying the Black Art*: Microsoft Press, 2006.

[3] H. Zhang and B. Kitchenham, "Semi-Quantitative Simulation Modeling of Software Engineering Process," in *Software Process Workshop/International Workshop on Software Process Simulation and Modeling (SPW/ProSim)* Shanghai: Springer, 2006.

[4] L. J. Osterweil, "Unifying Microprocess and Macroprocess Research," in *Software Process Workshop (SPW)* Beijing: Springer, 2005.

[5] E.-A. Karlsson, "Incremental Development - Terminology and Guidelines," in *Handbook of Software Engineering &Knowledge Engineering*. vol. 1, S.-K. Chang, Ed.: World Scientific, 2001.

[6] "QSIM," 4.3 Alpha 4 ed: UT Qualitative Reasoning Software, pp. http://www.cs.utexas.edu/users/qr/QR-software.html.

[7] B. Kuipers, *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*: MIT Press, 1994.

[8] T. K. Abdel-Hamid and S. E. Madnick, *Software Project Dynamics: An Integrated Approach*. Englewood Cliffs, N.J.: Prentice Hall, 1991.

[9] K. E. Huff, J. V. Sroka, and D. D. Struble, "Quantitative Models for Managing Software Development Processes," *Software Engineering Journal,* vol. 1, pp. 17-23, 1986.

[10] J. D. Tvedt, "An Extensive Model for Evaluating the Impact of Process Improvements on Software Development Cycle Time." vol. PhD: Arizona State University, 1996.

[11] D. Galin, *Software Quality Assurance: from Theory to Implementation*: Pearson, 2004.

[12] H. Zhang, M. Huo, B. Kitchenham, and R. Jeffery, "Qualitative Simulation Model for Software Engineering Process," in *17th Australian Software Engineering Conference (ASWEC)* Sydney: IEEE Computer Society, 2006.

[13] W. A. Florac and A. D. Careton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement*: Addison-Wesley, 1999.

[14] H. Zhang, B. Kitchenham, and R. Jeffery, "Planning Software Project Success with Semi-quantitative Reasoning," in *18th Australian Software Engineering Conference (ASWEC)* Melbourne, Australia: IEEE, 2007.

[15] H. Zhang, B. Kitchenham, and R. Jeffery, "A Framework for Adopting Software Process Simulation in CMMI Organizations," in *International Conference on Software Process (ICSP)* Minneapolis, MN: Springer, 2007.

[16] J. W. Grzymala-Busse, *Managing Uncertainty in Expert Systems*: Kluwer Academic Publishers, 1991.