UNIVERSITY OF CALGARY

Software Process Evaluation Using a Customizable Pattern-based Simulator

by

Keyvan Khosrovian Kermani

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

August, 2008

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Software Process Evaluation Using a Customizable Pattern-based Simulator" submitted by Keyvan Khosrovian Kermani in partial fulfillment of the requirements for the degree of Master of Science.

_____

Supervisor, Dr. Dietmar Pfahl,
Department of Electrical and Computer Engineering

_____

Co-supervisor, Dr. Vahid Garousi,
Department of Electrical and Computer Engineering

_____

Dr. Günther Ruhe,
Department of Electrical and Computer Engineering

_____

Dr. Mahmood Moussavi,
Department of Electrical and Computer Engineering

_____

Dr. Xin Wang,
Department of Geomatics Engineering

_____

Date

Abstract

Software process analysis and improvement relies heavily on empirical research. However, controlled experiments and surveys as means of empirical research have two major drawbacks. First, whatever evidence is gained via empirical research is strongly context dependent. Second, they are costly. Software process simulation modeling supports empirical studies by both reducing the cost of experimentation, and facilitating the combination of isolated empirical evidence. The process simulation model GENSIM 2.0, developed as part of this work, addresses the above challenges. Compared to the existing process simulation models in the literature, the novelty of GENSIM 2.0 is twofold: (1) Model structure is customizable to organization-specific development processes. This is achieved by using a limited set of generic structures (macro-patterns). (2) Model parameters can be easily calibrated to available empirical data and expert knowledge. This is achieved by making the internal model structures explicit and by providing guidance on how to calibrate the model parameters. The main achievements of the work are: Generic structures of software development processes referred to as macro-patterns, GENSIM 2.0, a customizable software process simulator developed based on the proposed macro-patterns and the V-Model lifecycle and complete and detailed description of the calibration of the simulator. This work also presents examples of useful application scenarios of the simulator including finding the best combination of Verification and Validation (V&V) techniques with respect to specific time, quality and effort goals and analyzing the effect of the project staffing profile on its performance.

Acknowledgements

My deepest gratitude goes to Dr. Dietmar Pfahl for his earnest and devoted support, encouragement and guidance throughout this work. Accomplishing this work would have been impossible without his supervision and assistance.

I am very grateful to Dr. Vahid Garousi, for supervision, providing valuable advices and sharing his experiences towards the end of this work.

I would also like to thank Dr. Gunther Ruhe, Dr. Mahmood Moussavi and Dr. Xin Wang, my committee members, for their help and feedbacks in revising this work.

Finally, I want to express my sincere appreciation to my drear cousins, Mehran and Rostam Pooladi-Darvish for their valuable support.

This work is dedicated to my dear parents.

Table of Contents

Approval Page.....ii
Abstract.....iii
Acknowledgements.....iv
Table of Contents.....vi
List of Figures.....viii
List of Tables.....ix
List of Equations.....xi

CHAPTER ONE: INTRODUCTION.....1
1.1 Software Process Improvement.....1
1.2 Motivation.....3
1.3 Limitations of previous work.....6
1.4 The proposed approach.....8
1.5 Contributions.....9
1.6 Thesis Outline.....10

CHAPTER TWO: BACKGROUND AND RELATED WORK.....11
2.1 System Dynamics.....11
2.1.1 Constructs of SD models.....11
2.2 System Archetypes.....13
2.3 Reusable Model Structures and Behaviours.....14

CHAPTER THREE: OVERVIEW OF GENSIM 2.0.....17
3.1 Generic Process Structures (Macro-Patterns).....17
3.2 The V-Model Development process.....21

CHAPTER FOUR: GENSIM 2.0 IMPLEMENTATION.....25
4.1 Model Parameters.....26
4.2 Views.....31
4.2.1 Development/Verification Views.....31
4.2.1.1 Code Phase Product Flow View.....32
4.2.1.2 Code Phase Defect Flow View.....34
4.2.1.3 Code Phase Resource Flow View.....36
4.2.1.4 Code Phase State Flow View.....39
4.2.2 Validation Views.....41
4.2.2.1 System Testing Product Flow View.....42
4.2.2.2 System Testing Defect Flow View.....44
4.2.2.3 System testing Resource Flow View.....45
4.2.2.4 System testing State Flow View.....47
4.3 Subscripts.....49
4.4 Workforce Allocation Algorithm.....51

CHAPTER FIVE: CALIBRATION OF GENSIM 2.0.....55
5.1 GENSIM 2.0 Calibration Parameters.....57

5.2 Sources of Calibration....59
5.3 Current Parameter Calibration of GENSIM 2.0....61
5.3.1 Calibration of the Development Phases....61
5.3.2 Calibration of Validation Phases....70

CHAPTER SIX: GENSIM 2.0 APPLICATION SCENARIOS....73
6.1 Scenario 1: Choosing the best combination of V&V techniques with respect to project performance goals....73
6.2 Scenario 2: Choosing the best combination of verification techniques with respect to project performance goals....78
6.3 Scenario 3: Analyzing the effect of workforce headcount on project performance....80
6.4 Scenario 4: Analyzing the effect of workforce skill levels on project performance....85

CHAPTER SEVEN: CONCLUSION AND FUTURE WORK....90
7.1 Contributions....90
7.2 Conclusion....91
7.3 Limitations and Future Work....92

REFERENCES....94
Appendix A....97
Appendix B....111

List of Figures

Figure 1: Schematic notation of a Rate and Level System 12

Figure 2: Example of the "limits to growth" system archetype [17] 14

Figure 3: Class Hierarchy of Reusable Model Structures [23] 15

Figure 4: Macro-pattern for development/verification activity pairs (with state-transition charts) 18

Figure 5: Test case development/Validation macro-pattern 21

Figure 6: Application of macro-patterns to simulate the V-Model in GENSIM 2.0 22

Figure 7: Code Phase Product Flow View 33

Figure 8: Code Phase Defect Flow View 35

Figure 9: Code Phase Resource Flow View 37

Figure 10: Code Phase State Flow View 40

Figure 11: System Test Product Flow View 42

Figure 12: System Test Defect Flow View 44

Figure 13: System Test Resource Flow View 46

Figure 14: System Test State Flow View 48

Figure 15: Quality vs. Duration and Effort vs. Duration (Scenario 1 – Calibrations A and B) 75

Figure 16: Quality vs. Duration and Effort vs. Duration (Scenario 2 – Calibration B) 78

List of Tables

Table 1: A subset of the parameters used in modeling the code phase 29

Table 2: A subset of the parameters used in modeling the system test phase 30

Table 3: Calibration parameters of the code development phase 57

Table 4: Calibration parameters of the system test phases 58

Table 5: Calibration values of the initial development parameters 61

Table 6: Calibration values of the correction effort parameters 65

Table 7: Calibration values of the artefact conversion parameters 65

Table 8: Calibration values of the fault conversion parameters 66

Table 9: Calibration values of the verification rates 66

Table 10: Defect Containment Matrix 67

Table 11: Total number of defects injected in different artefacts 68

Table 12: Calibration values of the defect injection parameters 69

Table 13: Calibration values of the verification effectiveness parameters 70

Table 14: Calibration values of the validation rate parameters 71

Table 15: Calibration values of the validation effectiveness parameters 71

Table 16: Difference between Calibration A and Calibration B 74

Table 17: Simulation results for Scenario 1 76

Table 18: Simulation results for Scenario 2 79

Table 19: Simulation results for Scenario 3 - Case 1 83

Table 20: Simulation results for Scenario 3 – Case 2 84

Table 21: Example of mapping skill levels from ordinal to ratio scale 86

Table 22: Differences in inputs of the runs of Scenario 4 – Case 1 87

Table 23: Simulation results for Scenario 4 – Case 1 87

Table 24: Workforce information for Scenario 4 – Case 2 88

Table 25: Simulation results for Scenario 4 – Case 2 89

List of Equations

Equation 1: Mathematical representation of a Rate and Level system 12

Equation 2: Skill level Matrix S 52

Equation 3: Portion of developers assigned to activity *j* in step *t* 54

Equation 4: Caculating the total initial code development duration 63

Equation 5: Calculating the initial code development productivity 63

Equation 6: Initial Staffing profile matrix for Scenario 3 - Case 1 81

Equation 7: Staffing profile matrix for run A of Scenario 3 - Case 1 82

Equation 8: Staffing profile matrix for run B of Scenario 3 - Case 1 82

Equation 9: Initial Staffing profile matrix for Scenario 3 – Case 2 84

Equation 10: Staffing profile matrix with arbitrary settings 85

Chapter One: Introduction

1.1 Software Process Improvement

Software industry has always been confronted with problems of overdue deadlines, cost overruns and poor quality of the products delivered by software development organizations. Meanwhile, the increasing demand for 'better, faster, cheaper' software together with the increasing complexity of software systems have urged software developers to bring discipline to software development processes and to improve the overall performance of software projects, i.e., shorter total project duration, less total project effort consumption (or cost), and better quality of the end product. In this context, a process is referred to as a set of activities, methods, practices, and tools that people use to develop and maintain a product and its associated work products (e.g., requirements specification, design documents, code and test cases) [1]. Two major factors affect the performance of software projects [2]. Firstly, there are the technological issues, e.g., languages, tools, hardware, etc. Despite of the considerable improvements achieved in these areas, experience shows that the extent to which these factors impact project performance is confined by human-related factors [3]. Secondly, there are the managerial issues such as planning, resource allocation and workforce training, that have a significant impact on project performance. However, advances in these areas are made with more difficulty due to the complex human-based nature of software development environments.

Empirical research is essential for developing theories of software process management, transforming the art of software development into an engineering discipline and

subsequently improving the overall performance of software development projects. Engineering disciplines require provision of evidence of the efficiency and effectiveness of tools and techniques in varying application contexts. In software engineering, the number of tools and techniques is constantly growing, and ever more contexts emerge in which a tool or technique might be applied. The application context of a tool or technique is defined, firstly, by the organizational aspects such as process structure, resource allocation, developer team size and skill sets, management policies, etc., and, secondly, by the set of all other tools and techniques applied in a development process.

Since most activities in software development are strongly human-based, the actual efficiency and effectiveness of a tool or technique can only be determined through real-world experiments. Controlled experiments and case studies are means for assessing the local efficiency and effectiveness of tools or techniques. Local efficiency and effectiveness of a tool or technique refers to the efficiency and effectiveness of the tool or technique when applied in isolation without considering its larger application context, for example, the typical defect detection effectiveness of an inspection or test technique applied to a specific type of development artefact, by a typical class of developers (with adequate training and experience levels) regardless of the other techniques and entities involved in a development process. Global efficiency and effectiveness of a tool or technique on the other hand, relates to its impact on the overall project performance while considering all other entities involved in the entire development process and their mutual influences. Typically, global efficiency and effectiveness have to be evaluated through a series of case studies.

1.2 Motivation

Controlled experiments and case studies are expensive in terms of effort and time consumption. Therefore, it is not possible to experiment with all practically relevant context alternatives in real projects. Hence, Support for deciding which experiments and case studies are more worthwhile to spend effort and time on would be a great help to focus empirical studies. Currently, these decisions are made purely expert-based, mostly relying on experience and intuition. This way of decision-making has two drawbacks. Firstly, due to the multitude of mutual influences between entities involved in a process, it is hard to estimate for an expert the extent to which a locally efficient and effective tool or technique positively complements another locally efficient and effective tool or technique applied in another activity of the chosen development process. Secondly, for the same reasons as in point one, it is hard to estimate for an expert how sensitive the overall project performance will react to variations in local efficiency or effectiveness of a single tool or technique. The second point is particularly important if a decision has to be made whether anticipated improvements are worthwhile to be empirically investigated within various contexts.

In order to assist decision makers in situations described above and to help minimize the drawbacks of the current decision making practice, one can provide experts with a software process simulation system that generates estimates of the impact of local process changes on the overall performance of the project. For example, if, derived from experiments with unit testing technique A, or from what its advocates claim, the defect detection effectiveness of unit testing technique A is locally 10% better than that of unit

testing technique B in a given context, through simulation we may find out that using technique A instead of technique B yields an overall positive impact of 2% on the end product quality (plus effects on project duration and effort) or we may find out the change yields an overall positive impact of 20%. If simulations indicate that it has only 2% overall impact or less, it might not be worthwhile to run additional experiments to explore the actual advantage of technique A over technique B (in the specific context).

With the help of simulation models, even more complex situations could be investigated. For example, one could assess the overall effectiveness and efficiency of different combinations of development, verification, and validation techniques. As an example, it is trivial that carrying out the best development activities and all the possible verification and validation techniques using highly skilled developers will result in good quality of the final product, but in a specific development context with specific resource constraints and deadlines, that may not be possible and the management would have to leave out some activities or invest in some activities more than others. In such a situation, the simulation model could be used to generate estimates of all project performance dimensions for all possible combinations of available development, verification and validation activities and help the management gain a better understanding of the 'big picture' of the development processes.

Another example of a complex situation where the simulation model could be helpful is a situation where management wants to investigate how much workforce should be allocated to development, verification and validation activities in order to achieve predefined performance goals. Obviously, in general, hiring a large workforce dedicated

to carrying out any of the activities within the development process will result in better performance regarding project duration. However, in a specific context, management might not be able to do so due to specific budget and resource constraints. In such a situation, the simulation model could help management in assessing different options of allocating workforce in order to stay within the time limits while not running out of available resources. More specifically, the simulation model enables management to get a better understanding of the overall development process with regards to different activities that are under progress at different points in time, and how workforce with skills in carrying out multiple activities could be used to meet the deadline while not running out of resources.

One can even go one step further and use process simulators to analyze how hiring more highly skilled developers or training the existing ones will affect the project performance. It is obvious that allocating workforce with high levels of experience with the tools and techniques that are used within the development context or investing in training the available workforce will generally result in better performance, but again, because of specific constraints, management might not always be able to do so. In a situation like this, a process simulator could be used to analyze the impact of the skill levels of different groups of workforce on the overall project performance by taking into account all specifics of the entire development process and assess whether and to what extent investments to increase workforce skills actually pay off.

1.3 Limitations of previous work

The idea of using software process simulators for predicting software project performance or evaluating software processes is not new. Beginning with pioneers like Abdel-Hamid [4], Bandinelli [5], Gruhn [6], Kellner [7], Scacchi [8], and many others, dozens of process simulation models have been developed for various purposes. However, all known models have at least one of the following shortcomings:

1. The model is too simplistic or its scope is so limited to actually capture the full complexity of real-world industrial development processes.
2. The model structure and calibration is not completely published and thus cannot be independently adapted and used by others.
3. The model captures a specific real-world development process with sufficient detail but fails to offer mechanisms to represent detailed product and resource models. This has typically been an issue for models using SD modeling environments.
4. The model structure captures a specific real-world development process (and associated products and resources) in sufficient detail, but is not easily adaptable to new application contexts due to lack of design for reuse and lack of guidance for re-calibration.

GENSIM [9], a predecessor of GENSIM 2.0, is an example of a model having the first shortcoming mentioned above. GENSIM is a software development process simulation model intended to be used for purely educational purposes. It models a basic waterfall-like development process with three phases of design, implementation and test, and mostly focuses on managerial dimensions related to the performance of the overall

software development project. Even though GENSIM is a good learning aid to familiarize software engineering students with managerial concepts and issues of software development projects, it has a simplified and limited scope making it unsuitable for more comprehensive analyses of development processes in real-world environments, where modeling and analyzing technical aspects of development as well as individual development phases and multiple project influences are critical.

An example of a model with the second shortcoming mentioned above is reported in [10]. The goal of building the model described in [10] is to facilitate quantitative assessment of financial benefits when applying Independent Verification and Validation (IV&V) techniques in software development projects and figuring out the optimal alternatives regarding those benefits. IV&V techniques are verification and validation techniques performed by one or more groups that are completely independent from the developers of a system and can be applied during all phases of the development. In this research, one NASA project using the IEEE 12207 [11] software development process with multiple possible IV&V configurations is modeled. The model is then used to answer multiple questions regarding application of IV&V activities in the software development project. Examples of these questions are: What would be the costs and benefits associated with implementing a given IV&V technique on a selected software project? How would employment of a particular combination of IV&V techniques affect the development phase of the project? Usefulness of the model is demonstrated using three different use cases. However, despite providing descriptions and snapshots of the overall structure, the implementation source of the model has not been made available to the public and therefore cannot be reused by others directly. This fact even limits the contributions of

the published experimental results, because the internal model mechanisms that generate the results cannot be evaluated by others.

In [12] Pfahl and Lebsanft report experience with a model having the fourth shortcoming mentioned above. The development and application of a process simulator called PSIM (Project SIMulator) was a pilot project conducted by Siemens Corporate Research within a Siemens business unit. Its purpose was to assess the feasibility of System Dynamics modeling and its benefits in planning, controlling and improving software development processes in a real-world environment. It modeled a development process comprising the high level design, low level design, implementation, unit test, and system test phases. Different available information sources were used to come up with the model structure and to calibrate the model parameters.

While the third shortcoming mentioned above can easily be resolved by fully exploiting the modeling constructs offered by commercial process simulation environments such as Extend™ [13] and Vensim® [14], the fourth issue has not yet been satisfactory resolved, neither by researchers proposing proprietary process simulation modeling environments (e.g., Little-Jil [15]) nor by researchers using commercial process simulation environments.

1.4 The proposed approach

Inspired by the idea of frameworks in software engineering, GENSIM 2.0 consists of a small set of generic reusable components which can be composed to model a wide range of different software development processes.

These components capture key attributes of different building blocks of different development processes. More specifically, they capture attributes of the product/process structure, and quality and resource specific attributes associated with the processes represented by the building blocks. Generally, product/process structure attributes relate mostly to the time dimension, quality attributes relate mostly to the quality dimension and resource attributes relate mostly to the effort dimension of project performance. However, what makes the model results interesting and hard to precisely predict are the numerous complex relationships and influences between each pair of these groups of attributes. GENSIM 2.0 currently assembles reusable building blocks, denominated macro-patterns and simulates an instance of the well-known V-Model software development process, consisting of three development phases (requirements specification, design and code) each comprising an artefact development activity and related verification activity (inspection) carried out on the developed artefact, and three validation (testing) activities, consisting of unit, integration and system test.

1.5 Contributions

The major contributions of this work are fourfold.

1. We introduce a limited set of generic process structures (macro-patterns) that can be reused in the development of software process simulation models and hence, improve the efficiency of these development activities.
2. In order to show the usefulness of the introduced macro-patterns, we developed a customizable software process simulator, GENSIM 2.0, based on these patterns. To enable easy reuse of the developed simulator we describe all its implementation details and equations in this work.

3. To facilitate easy re-calibration and reuse of GENSIM 2.0, we devote a complete chapter of this thesis to present a detailed description of all GENSIM 2.0 calibration parameters and how they are currently calibrated using data available in the software engineering literature. In addition, we discuss alternative sources that could potentially be used for the calibration of the model.
4. We describe example software process problems that GENSIM 2.0 can be applied to find a solution to. The presented application scenarios are not comprehensive but a subset of the variety of the situations that GENSIM 2.0 can be used in tackling software process issues.

1.6 Thesis Outline

This thesis is organized as follows. Chapter 2 presents related work. Chapter 3 presents an overview of GENSIM 2.0, the simulator that we have developed for the purpose of this thesis together with the generic process structures (macro-patterns). The implementation details of GENSIM 2.0 have been described in Chapter 4. In Chapter 5 the calibration of GENSIM 2.0 is discussed. Chapter 6 presents application scenarios of GENSIM 2.0 and finally we present the conclusions and future work in Chapter 7.

Chapter Two: Background and Related Work

2.1 System Dynamics

System Dynamics (SD) modeling was originally developed at MIT to solve socio-economic and socio-technical problems [16]. In its essence are the ideas of systems thinking [17]. In systems thinking, socio-economic or socio-technical systems are represented as feedback structures whose complex behaviours are a result of interactions of many (possibly non-linear) feedback loops over time [18]. During the past nearly 20 years, SD modeling has entered the software domain and has been used to analyze and tackle a wide range of issues regarding managerial aspects of software development projects [4], [19], [20], [21], [22]. Examples of its application in software engineering include the evaluation of process variants, project planning, control and improvement. In the next sub sections the process of building SD models and the constructs used in SD models are described.

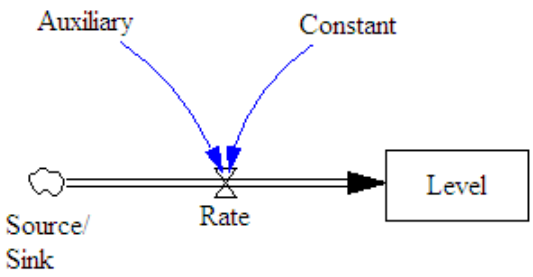
2.1.1 Constructs of SD models

The basic constructs used in SD modeling are levels, flows, sources/sinks, auxiliaries, constants and information links or connectors. Figure 1 depicts an example of a schematic representation of all these elements in a simple SD model implemented with Vensim®, a popular commercial tool used for SD modeling.

Level variables, also known as state variables, capture the state of the system by representing accumulations of entities. In the software engineering domain, level variables are used to represent accumulation of entities like software artefacts, defects and workforce.

Rate variables are always used together with level variables. They represent the flow of entities to or from the level variables. Example usages of rate variables in the software engineering domain are artefact development, defect generation and allocation of workforce.

Figure 1: Schematic notation of a Rate and Level System



Equation 1 shows how the value of a level variable is calculated at time $t + \Delta t$ using its value at time t. Level variables are in fact integrations of their input rates (inflows to the level) and output rates (outflows from the level) over time.

Equation 1: Mathematical representation of a Rate and Level system

$$Level(t + \Delta t) = Level(t) + \left(\sum_{over\ all\ inputs} Input\ \ Rate(t) - \sum_{over\ all\ outputs} Output\ \ Rate(t) \right) \Delta t$$

Sources and Sinks represent the system borders. Typically entities leaving the system are sent to sinks, e.g., software artefacts delivered to the customer and entities from outside the boundaries of the system enter the system from sources, e.g., newly hired workforce.

Auxiliaries are variables that are used for intermediate calculations, e.g., the portion of a software artefact that needs to be reworked due to defects detected during unit test.

Constants are used to represent factors that determine the modeled system. In other words, they are means for calibrating the simulation model to its context. Constants keep their initial value during the simulation. The average number of errors that developers commit while developing a kind of software artefact is an example of a constant.

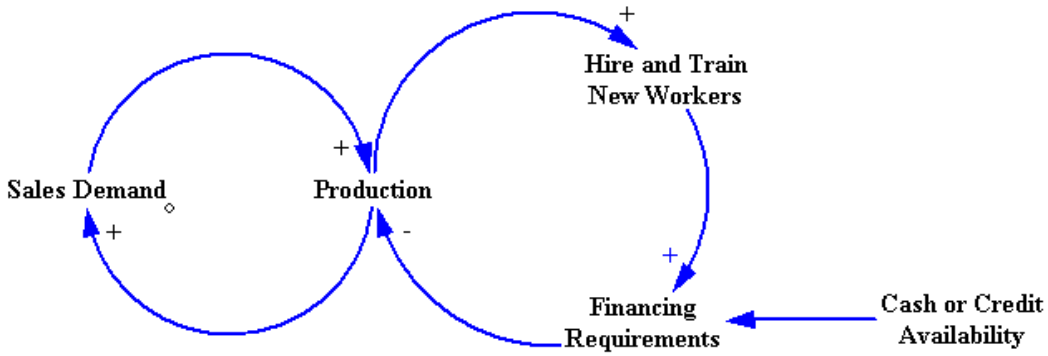
Information Links or connectors represent flow of information. When one variable is connected to another, this means that the value of the former has an influence on the value of the latter. For example, in Figure 1, the values of the Auxiliary and the Constant are used to calculate the value of the Rate.

2.2 System Archetypes

A first attempt to define a set of core structures of process simulation models which can be seen as a set of basic building blocks of any process simulator was made by Peter Senge in the early 1990s [17]. He identified ten "Systems Archetypes", i.e., generic process structures which embody typical recurring behavioural patterns of individuals and organizations. "Limits to growth" is one of these archetypes which he explains as "A reinforcing (amplifying) process is set in motion to produce a desired result. It creates a spiral of success but also creates inadvertent secondary effects...which eventually slow down the success". Figure 2 shows an example of this behavior in an organization.

As can be seen, growth in sales demand in a certain production organization leads to a growth in its production. Meanwhile, growth in its production leads to growth in its sales demand as well. This simple mutual effect could be considered as a "spiral of success". However, on the other hand, growth in production requires hiring and training new workforce.

Figure 2: Example of the "limits to growth" system archetype [17]

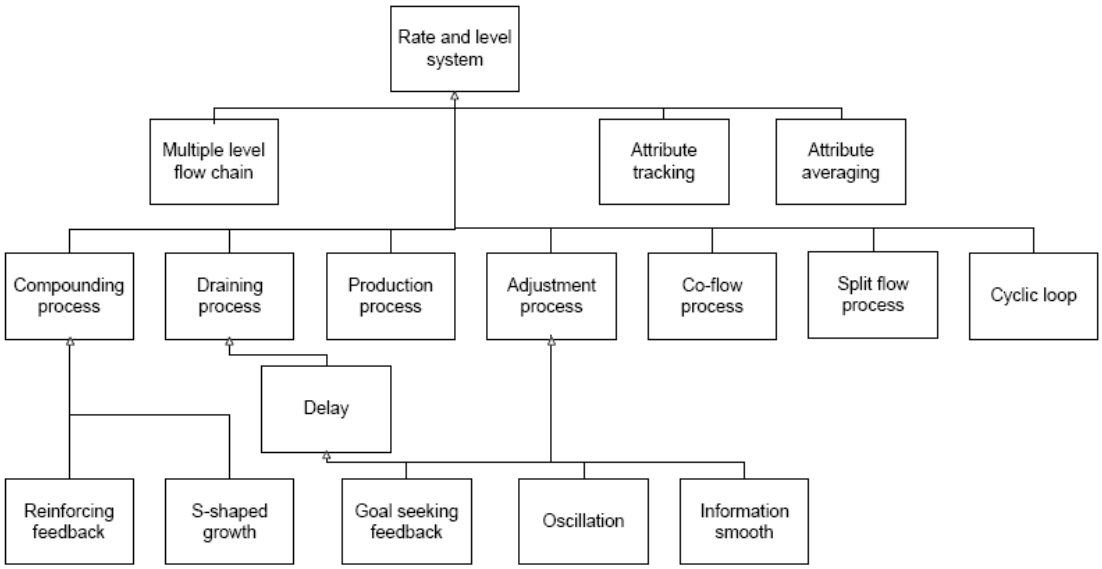


In this example, the availability of new workforce and limited financing capacity are among the factors that "slow down the success". Although these archetypes are certainly a good tool for understanding individual and organizational behaviour modes, they are too generic and qualitative as to be directly applicable for the modeling of software development processes.

2.3 Reusable Model Structures and Behaviours

More recently, following the approach taken by Senge but having software development processes in mind, Raymond Madachy suggested a core set of reusable model structures and behaviour patterns [23]. His proposed set comprises several very specific micro-patterns (and their implementations) suited for System Dynamics process simulation models. The object-oriented framework concept has been used for organizing these structures in a class hierarchy with inheritance relationships in which as you move down the hierarchy the structures become bigger and more complex as shown in Figure 3.

Figure 3: Class Hierarchy of Reusable Model Structures [23]



At the root of the hierarchy is the rate and level system, the smallest individual class in a System Dynamics model. Instances of this class are referred to as elements. Below the root class are classes representing generic flow processes. Generic flow processes are small microstructures consisting of only a few elements. Underneath the classes representing generic flow processes are infrastructures that comprise several microstructures producing more complex behaviours. Finally, there are the flow chains that form the basic "backbone" of a model portion. These flow chains are larger infrastructures including a series of elements. This set of reusable structures or "plug and play" components can be put together to build System Dynamics models for software development processes of varying complexity.

Madachy's micro-patterns are well-thought reusable process structures, with very specific purpose and focused scope. They can be interpreted as a bottom-up approach to support

reusability of process simulation structure. However, there exist no guidelines that help modellers combine individual micro-patterns to capture more complex, software development specific process structures.

Emerging from suggestions made several years ago [24], the work presented in this thesis complements Madachy's micro-patterns by a top-down approach that provides a set of reusable and adaptable macro-patterns of software development processes. The suggested macro-patterns are described in more detail by giving an implementation example of the research prototype GENSIM 2.0. Besides capturing important structural and behavioural aspects of software development processes, GENSIM 2.0 provides a blueprint on how to integrate detailed product and resource models.

Chapter Three: Overview of GENSIM 2.0

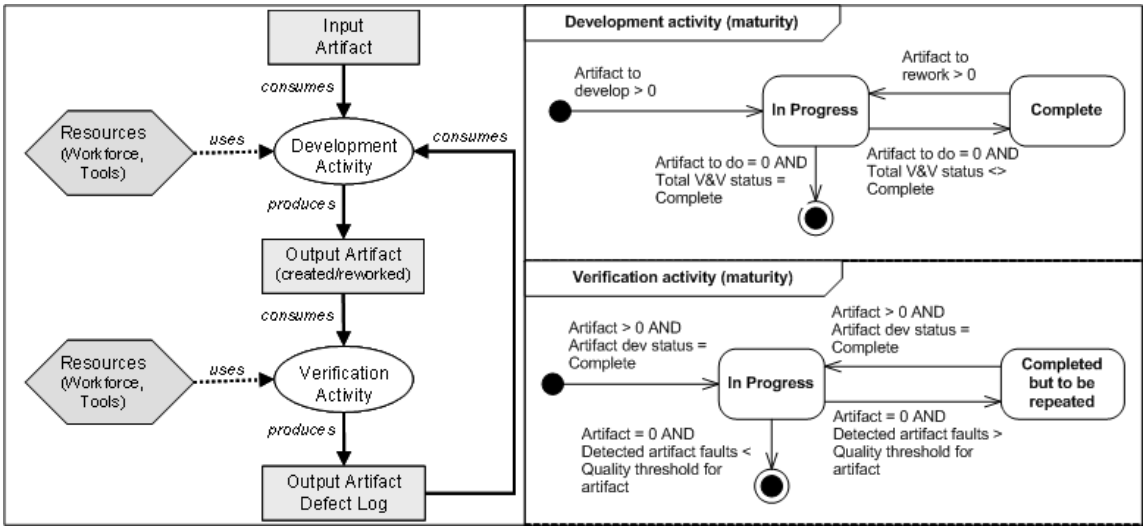
Inspired by the idea of frameworks in software engineering, customizable software process simulation models and frameworks can be constructed using generic and reusable structures ([25], [26], [27]) referred to as macro-patterns. GENSIM 2.0 is an example of a process simulation model constructed from macro-patterns. This chapter describes the macro-patterns of software development processes as employed in GENSIM 2.0. The design of the macro-patterns used for constructing GENSIM 2.0 is derived from generic process structures that are common in software development.

3.1 Generic Process Structures (Macro-Patterns)

The left-hand side of Figure 4 illustrates the macro-pattern that GENSIM 2.0 employs for development activities (comprising initial development and rework) and its associated verification activities. As shown in the figure, it is assumed that software artefacts (requirements specification, design and code) are verified (e.g., inspected) right after they are developed. However, since not in all software development projects all artefacts are verified, this activity is optional.

Associated with activities, are input/output products and resources. It is assumed that every development activity has some input artefacts and cannot be started if those artefacts are not ready. For example, the code development activity may not be started without the related design artefacts in place. Outputs of the development activities which are software artefacts are the input for the verification activities. Output of the verification activities are defects logs which are fed back to the development activities for reworking of the software artefacts.

Figure 4: Macro-pattern for development/verification activity pairs (with state-transition charts)



In addition, each artefact, activity, and resource is characterized by attributes representing states. *Learning* is an example attribute related to resources such as workforce. The number of times an activity has been carried out may be used to determine the learning state. Other states may represent the maturity of activities. The right-hand side of Figure 4 shows the state-transition diagrams determining the maturity states of development (top) and verification (bottom) activities.

In the state transition diagram of the development (initial development or rework) activity it can be seen that as soon as the target size of the artefact that has to be developed becomes greater than zero, i.e., input artefacts of the development activity are ready and development can be started, the development activity transitions into the *In Progress* state. After development of an artefact is finished, if verification has to be carried out, the artefact is handed to the verification team and the development activity transitions into

the *Complete* state. The same transition happens when rework of artefacts is finished and they are handed to validation teams for testing activities. Hence, in the diagram, state transitions of the development activity are specified using the *Total V&V status* which represents the state of all V&V (Verification and Validation) activities together. After the verification activity is finished, the development activity transitions into the *In Progress* state again as the artefact has to be reworked. This transition happens similarly in the situation where validation activities are finished and artefacts have to be reworked as a result. Whenever the development activity of an artefact is finished and no more verification and validation has to be carried out the development activity of the artefact is finalized.

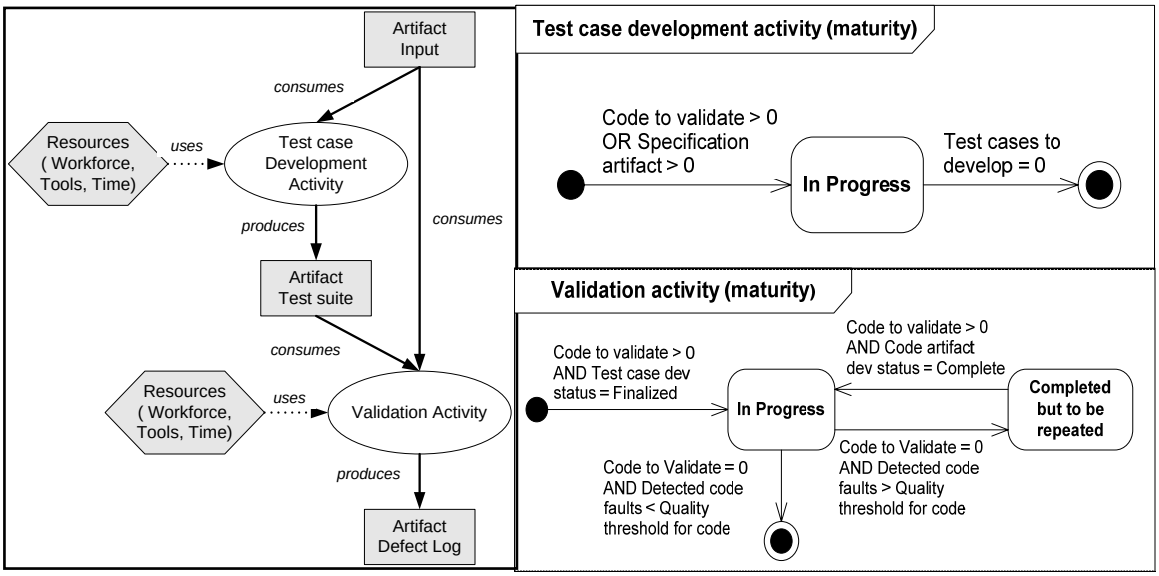
In the state transition diagram of the verification activity it can be seen that the verification activity transitions into the *In Progress* state as soon as the development activity of an artefact is finished. Whenever the verification activity is finished, depending on the number of detected defects, it is either finalized or goes into the *Complete but to be repeated* state. If the number of defects that is detected is below a certain threshold, the verification activity is finalized; otherwise it goes into the *Complete but to be repeated* state to indicate that due to great number of detected defects the artefact has to be verified once more. However, since this policy, which implies the enforcement of quality thresholds, might not be followed in all organizations, it is optional. If thresholds are not used, every verification activity is carried out at most once.

The Left-hand side of figure 5 illustrates the macro-pattern applied for validation phases of the development process. In the figure it can be seen that it is assumed that certain artefacts, i.e., software code or specification artefacts, depending on management

policies, are input to any test case development activity. The test case development activity cannot begin if these artefacts are not in place. Output of the test case development activity is a collection of test cases in the form of a test suite. If test case verification activity has to be modeled, the test case development activity can be extended to include both the test case development and verification activities using the development/verification macro-pattern explained above. The developed test suite along with other necessary artefacts, i.e., software code artefacts, is the input to the validation activity. The output of the validation activity is a log of all detected defects which is fed back to the development activity for rework. Test case development and validation activities, like any other activity, use resources.

The right-hand side of figure 5 shows the state transition diagrams specifying the maturity states of the test case development (top) and validation (bottom) activities. The test case development activity transitions into the *In Progress* state whenever software code or specification artefacts are available. Determining whether or not test cases can be derived directly from the specification artefacts before the code artefacts are ready, i.e., *Specification artefact* is greater than zero while *Code to validate* is still zero depends on managerial policies and the nature of the specific testing activity itself. Whenever there are no more test cases to develop in order to test the code artefacts the test case development activity is finalized.

Figure 5: Test case development/Validation macro-pattern

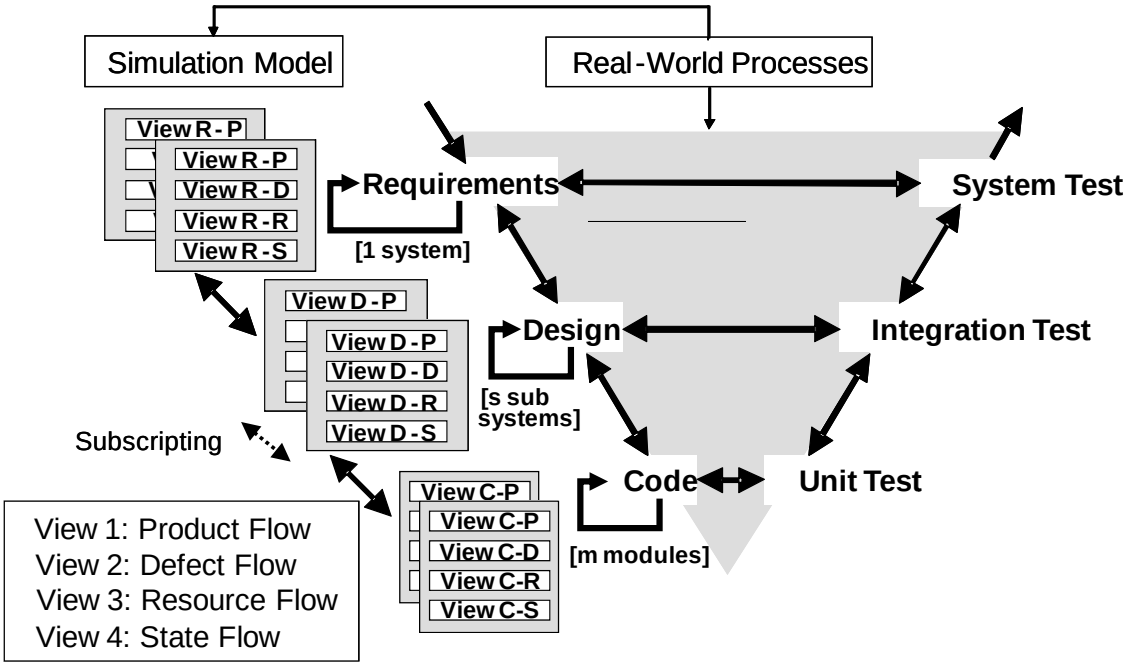


The code artefact validation activity transitions into the *In Progress* state whenever code artefacts that have to be validated are available and the required test cases are developed. Whenever all of the code artefacts are tested, depending on the number of detected defects, the validation activity is either finalized or transitions into the *Completed but to be repeated* state. If the number of detected defects is lower than a certain threshold, the activity is finalized. If it is greater than the threshold, it transitions into the *Completed but to be repeated* state showing that the artefacts have to be re-tested. From the *Completed but to be repeated* state, the validation activity transitions into the *In Progress* state as soon as reworking of the artefacts is finished and they become available for validation.

3.2 The V-Model Development process

For the current implementation of GENSIM 2.0, the macro-patterns shown in figures 4 and 5 are employed to represent an instance of the well-known V-Model software development process shown in figure 6.

Figure 6: Application of macro-patterns to simulate the V-Model in GENSIM 2.0



As described in the development/verification macro-pattern, every development activity is immediately followed by a verification activity. In the figure this is shown using the loops from the development phases to themselves. In GENSIM 2.0, different software artefacts are captured as instances of different software artefacts types. For example, a specific design artefact of a subsystem is an instance of the artefact type *design artefact*. Software artefacts types are associated with one of three granularity or refinement levels of software development as follows:

System Level includes activities carried out on artefacts representing the whole system. It currently consists of the requirements specification development and verification (e.g., requirements specification inspection) pair and the system testing activities.

Subsystem Level includes activities carried out on artefacts representing individual subsystems. It currently consists of design development and verification (e.g., design inspection) pair and the integration testing activities.

Module Level includes activities carried out on artefacts representing individual modules. It currently consists of code development and verification (e.g., code inspection) pair and the unit testing activities.

On each level, one or more artefacts are developed, verified, and validated. If a new refinement level is required, e.g., design shall be split into high-level design and low-level design, existing models can easily be reused to define separate high-level and low-level design levels replacing the current subsystem level.

Only development activities are mandatory. Depending on the organizational policies, verification and validation activities might not be performed. Therefore, all V&V activities are made optional. If defects are detected during verification or validation, rework has to be done. On code level, rework is assumed to be mandatory regardless of the activity that has detected the defects. Rework of design and requirements artefacts is optional for defects found by verification and validation activities of subsequent phases.

In order to capture the main dimensions of project performance, i.e., project duration, project effort, and product quality, and to explicitly represent the states of activities, the software development process shown in Figure 6 is implemented in separate views, each view representing one of the following four dimensions of each development/rework & verification macro-pattern and each validation macro-pattern:

Product Flow View models the specifics of how software artefacts are processed (developed, reworked, verified and validated) and sent back and forth during and between different activities of the development project.

Defect Flow View models the specifics of how defects are moved around (generated, propagated, detected and corrected) as different software artefact are processed (as modeled in the product flow view). In other words, it is a co-flow of the product flow and captures the changes in the quality of different software artefacts as they are processed in different activities.

Resource Flow View models the specifics of how different resources (developers, techniques/tools) are allocated to different activities of the development project.

State Flow View models the specifics of how the states of different entities as explained in Section 3.1 change during the development project.

As mentioned earlier, different software artefacts types are associated with refinement levels of the software development process, i.e., system, subsystem, and module. In the implementation of GENSIM 2.0 the subscripting mechanism provided by Vensim® has been used to model individual software artefacts. Therefore if one system consists of several sub-systems, and each sub-system of several modules, then each of the individual software artefacts belonging to any of the subsystems or modules is identifiable using the subsystem's or module's subscript value.

Chapter Four: GENSIM 2.0 Implementation

GENSIM 2.0 is implemented using the System Dynamics (SD) simulation modeling tool Vensim®, a mature commercial tool widely used by SD modelers. Vensim® offers three features in support of reuse and interoperability: views, subscripts, and the capability of working with external Dynamic Linked Libraries (DLL).

The capability of Vensim® to have multiple views is used to capture the main dimensions of project performance (i.e. project duration, project effort, and product quality), as well as the states of the software development process shown in Figure 6. Having multiple views adds to the understandability and hence reusability of the model, while enabling the modeller to focus on one specific aspect at a time. Views are discussed in more detail in Section 4.2.

The subscripting mechanism provided by Vensim® is used to model individual software artefacts. This again adds to the reusability of the model because the model can be easily reused to simulate different numbers of individual products in different projects. For example, if the model is used to simulate a development project for a software product consisting of five subsystems, the model can be easily reused to simulate a project for a software product that has six subsystems by simply changing the *Subsystem* subscript range from five to six. Besides reusability, application of subscripts adds to the level of detail that the model can capture since it can capture individual entities. Subscripts and their usage in GENSIM 2.0 are discussed in more detail in Section 4.3.

The capability of Vensim® to work with external DLLs is used to extract organization-specific heuristics from the SD model and incorporating them into external DLL libraries where they can be modified easily without affecting the model structure. An example of

such a heuristic is the workforce allocation algorithm or policy. The possibility to extract computation intensive heuristics from the process simulation adds to the customizability and reusability of GENSIM 2.0, since different organizations potentially have different policies to allocate their available workforce to different tasks. A process simulation model that hard-wires one specific allocation algorithm has to be modified extensively in order to be reused in another organization. These issues are discussed in more detail in Section 4.4.

4.1 Model Parameters

GENSIM 2.0 has a large number of parameters. Input and calibration parameters are typically represented by model constants, while output parameters can be any type of variable, i.e., levels, rates or auxiliaries. Generally, since the macro-pattern described in Section 3.1 is employed in modeling all the three development phases, corresponding sets of parameters have been defined for each development phase. To give an example, in the code phase model, the level variable *Code to do size* is used to represent the amount of code artefact that is waiting to be developed. Meanwhile, the level variable called *Design to do size* is defined in the design phase model to specify the amount of a design artefact that is waiting to be developed. The same rule applies for the set of parameters used in modeling different validation phases. This mechanism, adds to the understandability and hence reusability of the model.

Parameters can represent model inputs and outputs, or they are used to calibrate the model to expert knowledge and empirical data specific to an organization, process, technique or tool. Table 1 shows a subset of the parameters used in the implementation of the code phase (comprising code development and verification activities). Corresponding

parameters exist for the requirements specification and design related sub-processes. These parameters and the influences between them are discussed in more detail in Section 4.2.1. For a complete description and the details of all the parameters and their defining equations please refer to Appendix A.

Input parameters represent project specific information such as estimated product sizes, developer skills and project specific policies such as the combination of the verification and validation activities or whether requirements or design artefacts should be reworked if defects are found in code that actually originate in requirements specification or design phases.

Calibration parameters represent organization specific information that is typically retrieved from measurement programs and empirical studies. For a detailed description of how GENSIM 2.0 calibration is done, refer to Chapter 5.

Output parameters represent values that are calculated by the simulation engine based on the dynamic cause-effect relationships between input and calibration parameters. Which output values are in the focus of interest, depends on the simulation goal. Typically, project performance variables such as product quality (e.g., in terms of total number of defects or defect density), project duration (e.g., in terms of calendar days) and effort consumption (e.g., in terms of person-days) are of interest.

Besides their usage in the simulation model, i.e., input, calibration and output, the parameters within GENSIM 2.0 can also be categorized according to the entity which they represent an attribute of. In GENSIM 2.0 it is assumed that different parameters can be an attribute of four different entities namely process, product, resource and project.

Attributes of the process category define the structure of the development process or the specifics of how different activities are carried out, e.g., *verify code or not*. The *verify code or not* parameter is a Boolean constant that specifies if the code verification activity is carried out or not. This directly affects the process structure.

Attributes of the product category define the specifics of the software product that is being developed e.g., *number of modules per subsystem*, which specifies the number of modules within different subsystems of the software product.

Attributes of the resource category capture the specifics of the available resources for the project including tools/techniques and the workforce e.g., *Developers' skill level for code dev* and *Maximum code ver effectiveness*. The *Developers' skill level for code dev* parameter is a constant that defines the skill level of the available workforce in developing code artefacts. The *Maximum code ver effectiveness* parameter is a constant that defines the effectiveness of the code verification tool/technique in detecting code faults in the code artefacts.

The last group of parameters is the one that relates to the attributes of the overall project. It mostly captures the software development context and managerial policies. For example, *Required skill level for code dev* is a constant that represents the management policy regarding the skill level of the workforce that can be allocated to carry out code development tasks.

Parameters of GENSIM 2.0 can also be classified according to the view that they are associated with. Which type of view, i.e., product, defect, resource or state flow view a parameter is associated with depends on the primary effect of the attribute it represents.

For example, *Required skill level for code dev* is a parameter representing a managerial policy that primarily affects the resource flow of the code development activity.

Table 1: A subset of the parameters used in modeling the code phase

No.	Parameter Name	Type	Attribute of	View
1	Verify code or not	Input	Process	C-P
2	# of modules per subsystem	Input	Product	C-P
3	Developers' skill levels for code dev	Input	Resource	C-R
4	Developers' skill levels for code ver	Input	Resource	C-R
5	Code doc quality threshold per size unit	Input	Project	C-S
6	Required skill level for code dev	Input	Project	C-R
7	Required skill level for code ver	Input	Project	C-R
8	Code rework effort for code faults detected in CI	Calibrated	Process	C-D
9	Code rework effort for code faults detected in UT	Calibrated	Process	C-D
10	Code rework effort for code faults detected in IT	Calibrated	Process	C-D
11	Code rework effort for code faults detected in ST	Calibrated	Process	C-D
12	Average design to code conversion factor	Calibrated	Product	C-P
13	Average # of UT test cases per code size unit	Calibrated	Product	C-P
14	Average design to code fault multiplier	Calibrated	Product	C-D
15	Maximum code ver. effectiveness	Calibrated	Resource	C-D
16	Maximum code ver. rate per person per day	Calibrated	Resource	C-P
17	Initial code dev. rate per person per day	Calibrated	Resource	C-R
18	Minimum code fault injection rate per size unit	Calibrated	Resource	C-D
19	Code to rework	Output	Process	C-P
20	Code development activity	Output	Process	C-P
21	Code verification activity	Output	Process	C-P
22	Code development effort	Output	Process	C-R
23	Code verification effort	Output	Process	C-R
24	Code faults undetected	Output	Product	C-D
25	Code faults detected	Output	Product	C-D
26	Code faults corrected	Output	Product	C-D
27	Code doc size	Output	Product	C-P

Table 2 shows a subset of the parameters used in the implementation of the system test phase. Corresponding parameters exist for the unit and integration test phases. These parameters and the influences between them are discussed in more detail in Section 4.2.2. For a complete description of the details of all the parameters and equations refer to Appendix A.

Table 2: A subset of the parameters used in modeling the system test phase

No.	Parameter Name	Type	Attribute of	View
1	System test or not	Input	Process	S-P
2	Postpone TC dev until code is ready in ST or not	input	Process	S-P
3	Developers' skill levels for ST	Input	Resource	S-R
4	Quality threshold in ST	Input	Project	S-S
5	Required skill level for system test	Input	Project	S-R
6	Maximum ST effectiveness	Calibrated	Resource	S-D
7	Maximum ST productivity per person per day	Calibrated	Resource	S-P
8	Maximum # of ST test cases developed per person per day	Calibrated	Resource	S-P
9	Maximum # of ST test cases executed per person per day	Calibrated	Resource	S-P
10	Number of test cases for ST	Calibrated	Product	S-P
11	Code returned for rework from ST	Output	Process	S-P
12	ST rate	Output	Process	S-P
13	Incoming code to ST rate	Output	Process	S-P
14	System testing effort	Output	Process	S-P
15	Code ready for ST	Output	Product	S-P
16	ST test cases	Output	Product	S-P
17	Actual code faults detected in ST	Output	Product	S-D

Thanks to the *gaming* feature provided by Vensim®, the value of any GENSIM 2.0 input and calibration parameter could be changed in the middle of a simulation run. In terms of the impacts of a change in a parameter value on the model, we can group the parameters in two categories: (1) the parameters which are only read (i.e., used) at a single designated time step only and any change in those parameters after that time step will simply not be considered by the model, and (2) parameters which are read in every time step and, thus, any change in those at any time step will be considered by the model.

As an example of case (1) above, any change in the average size of the requirements specification artefact made after the first time step, will not be considered by the simulator as this parameter is read and used only at the beginning of the simulation.

As an example of case (2) above, changing the average development productivity parameters, denoting the real situation in an organization in which the average productivity is increased or decreased on a given day or week, will always be considered as these parameters are read in every time step.

4.2 Views

In this section the implementation of the four views mentioned above and their underlying assumptions and internal mechanisms are described for both the development phases (development/verification activities) and validation phases in more detail.

4.2.1 Development/Verification Views

In this section, underlying assumptions and mechanisms, levels, rates, and auxiliary variables implemented in the four different views of the development phases of the development project, i.e., requirements specification, design and code as illustrated in Figure 6 are discussed in more detail. Since the macro-pattern discussed in Section 3.1 is applied to product flow views of all the three development phases (i.e. requirements specification, design and code), all the four views are similar for all of them except for few minor differences related to the specific nature of the development phase. For example, in the code phase product flow view, three rate variables are defined to represent the outflow of code artefacts to other phases, i.e., the three validation phases. However, in the design phase product flow view, only one rate variable is defined to represent the outflow of design artefacts to other phases, i.e., the code phase. In the following, only the code phase is explained in full detail.

4.2.1.1 Code Phase Product Flow View

The code phase product flow view captures the specifics of how code artefacts are developed, reworked and verified and sent back and forth during and between the code phase and validation phases of the development project. Figure 7 is a simplified snapshot of the code phase product flow view with many of its auxiliary variables hidden to improve readability and understandability of the graph.

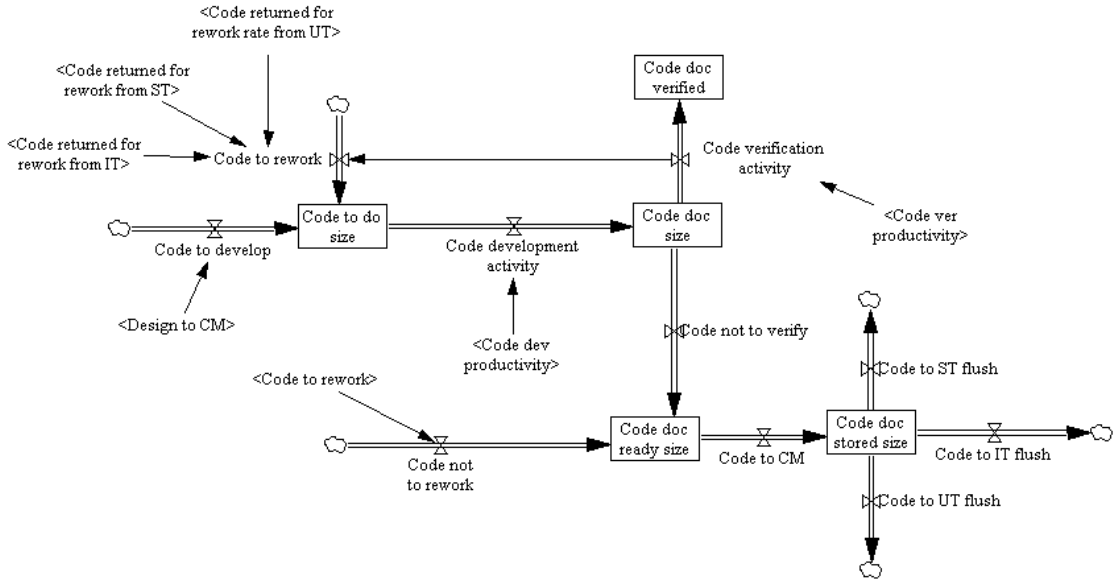
Software artefacts that flow through this part of the model are code artefacts of different modules of the system. It is assumed that code development for a module can only begin when the design artefacts for the subsystem that the module belongs to is completed. This is specified in the model with the information link from the *Design to CM* (Configuration Management), which is itself a rate variable in the design development/verification product flow view, to the *Code to develop* rate. Variables in the form of <…> define the interface of this view to other views. *Code to develop* rate is a variable which specifies the incoming flow of code artefacts that has to be developed. These artefacts are stored and wait in the *Code to do size* level variable before they can be developed. As soon as *Code dev productivity* becomes greater than zero, i.e., developers become available to carry out the code development task, waiting code artefacts are developed and then stored in the *Code doc size* level variable.

Whenever the development activity for a module's code artefact is finished, it is either verified or not according to the state variables as discussed in Section 4.2.1.4. If the code artefacts have to be verified they have to wait in the *Code doc size* level variable until the *Code ver productivity* becomes greater than zero, i.e. verifiers become available and can carry out the verification task. While the *Code verification activity* is greater than zero,

i.e. the code verification activity is under process, the *Code doc verified* level variable is used to keep track of the amount of code that has been verified at any moment.

As code is verified and code faults are detected in the code artefact, the code artefact is sent back for rework using the *Code to rework* rate variable. This rate is also used to specify the amount of code artefact returned for rework from the validation phases (i.e. unit, integration and system test).

Figure 7: Code Phase Product Flow View



If at the end of the development activity the code artefact doesn't need to be verified, it flows to the *Code doc ready size* level variable using the *Code not to verify* rate variable. The *Code not to rework* rate variable is needed, because in some situations only parts of the code artefact have to be sent for rework. These situations are the times when few, i.e., less than a certain a threshold code faults are detected and reworking the entire code

artefact is not necessary. The parts that do not need rework flow to the *Code doc ready size* level variable using the *Code not to rework* variable.

When all parts of a module's code artefact arrive in the *Code doc ready size* level variable they are stored in the *Code doc stored size* level variable using the *Code to CM* rate variable. The *Code doc stored size* corresponds to the configuration managements system. *Code to UT flush, Code to IT flush and Code to ST flush* rate variables are used for sending the code artefact to different validation phases.

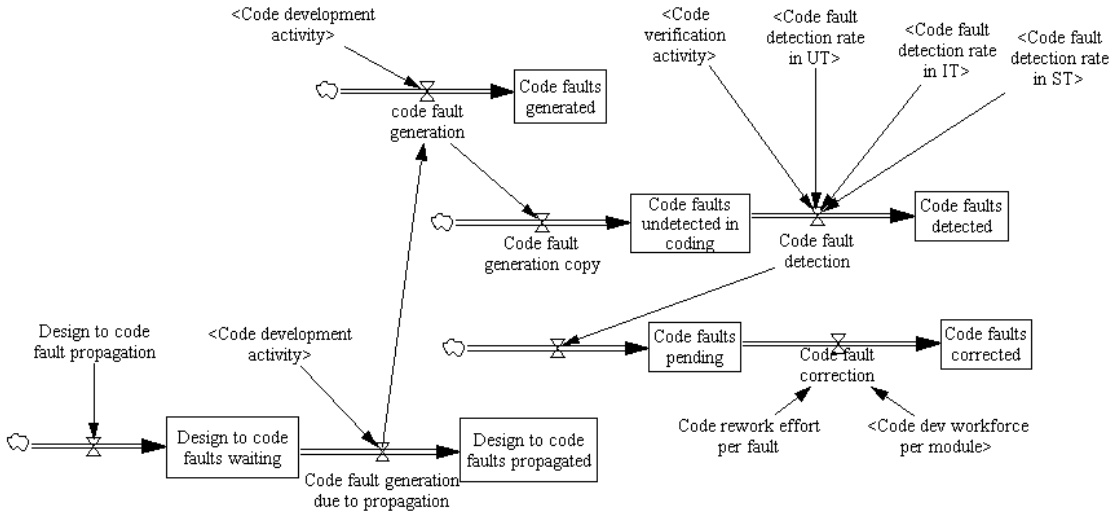
4.2.1.2 Code Phase Defect Flow View

This view captures the specifics of how defects are moved around i.e. generated, propagated, detected and corrected as code artefacts are processed (as modeled in the code phase product flow view). In other words, it is a co-flow of the code phase product flow and captures the changes in the quality of code artefacts as they are processed in the code phase. Figure 8 shows a simplified snapshot of this view with many of its auxiliary variables hidden to improve readability and understandability of the graph.

Entities that flow through this view are code faults that exist in the code. It is assumed that code faults are injected in the code artefact for two reasons. Firstly, there are the faults that are injected in the code artefact due to design faults in the design artefact that have not been detected and hence have propagated into the coding phase. These faults are specified using the *Design to code fault propagation* variable. These faults will not be injected into the code unless the code development activity begins. Therefore, they are stored in the *Design to code fault waiting* level variable and wait there until the *Code development activity* becomes greater than zero and hence causing these faults to be actually injected into the code artefact using the *Code fault generation due to*

propagation rate variable. The *Design to code faults propagated* level variable is used to keep track of the number of faults that has been committed in the code artefact because of the propagation.

Figure 8: Code Phase Defect Flow View



The second group of faults that are injected into the code are the ones due to mistakes made by the developers. These faults are specified using the information link from the *Code development activity* to the *Code fault generation* rate variable. *Code fault generation* specifies the sum of faults committed with both of the sources.

The generated faults are stored in the *Code faults undetected in coding* level variable and wait until some of them are detected due to verification and validation activities. The final remaining undetected faults will be the ones that will remain in the code after shipment of the product. The *Code fault detection* rate variable specifies the sum of code

faults detected in various V&V activities. The *code faults detected* level variable is used to keep track of the number of code faults that are detected.

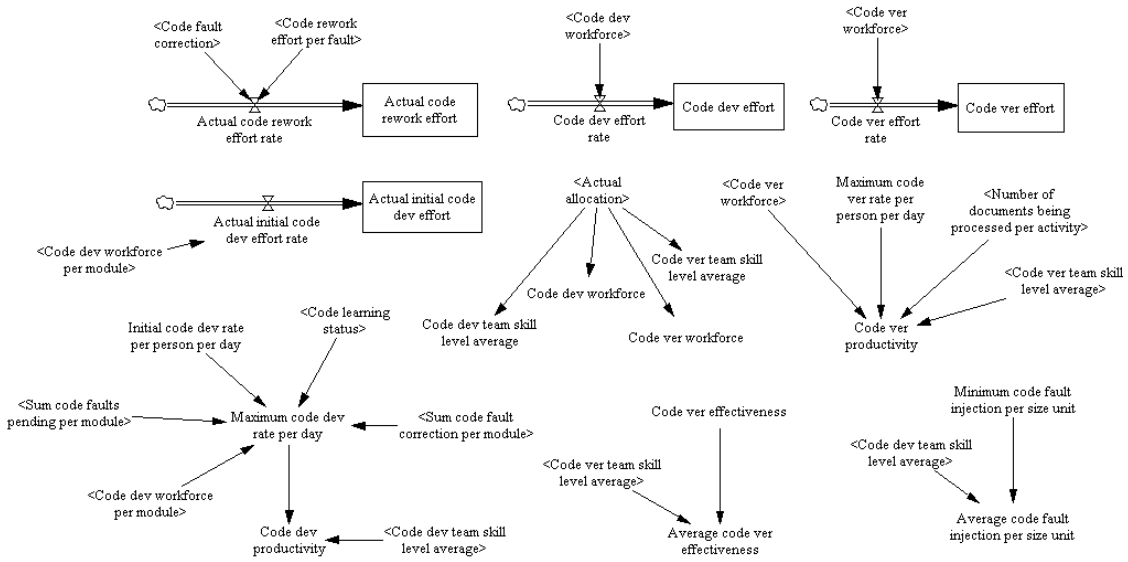
After code faults are detected, they are stored in the *Code faults pending* level variable where they wait until they are fixed. It is currently assumed that all of them are corrected during rework. The *Code faults correction* rate specifies the number of code faults that are corrected per time unit. The rate depends on the headcount of workforce that are reworking the code artefact and the amount of effort that has to be spent to fix each of the faults. The C*ode faults corrected* level variable is used to keep track of the number of code faults that have been fixed during the reworking of the code artefacts.

4.2.1.3 Code Phase Resource Flow View

This view captures various attributes related to resources, i.e., developers (workforce) and techniques/tools that are used to perform the code phase activities of the development project. Figure 9 depicts a simplified snapshot of this view with some of its auxiliary variables hidden to improve readability and understandability of the graph.

The *Actual Allocation* is an important auxiliary variable that uses the external DLL library of GENSIM 2.0. In essence, it is a matrix consisting of one row for any of the activities within the project and two columns. The first column represents the headcount of the workforce allocated to the activities. The second column represents the average skill level of the team allocated to the activity. As can be seen in Figure 9, it is used to determine the headcount of the workforce assigned to the code development and verification activities and their skill level average. Details on exactly how *Actual Allocation* is calculated are discussed in Section 4.4.

Figure 9: Code Phase Resource Flow View



It is assumed that the skill level of an employee is specified by a real number between 0 and 1, where 0 means "not able to carry out the activity" and 1 means "optimally skilled". If such exact information can not be specified, but the data can be given on an ordinal scale a mapping from the ordinal scale onto [1,0] could resolve the issue (Details are discussed in Chapter 6).

Code ver effectiveness is a constant used to represent the effectiveness of the code verification technique in detecting the code faults in the code artefact, if used by "optimally skilled" workforce. It has a value between 0 and 1. If for example effectiveness of a certain code verification technique is 0.7, it means that when using the technique 70% of the faults in the code will be detected. If the skill level average of the assigned workforce is less than "optimally skilled", the value of this variable decreases

proportionately. This constant has to be calibrated based on information about the training and experience of the workforce.

Minimum code fault injection per size unit is a constant used to represent the number of faults that "optimally skilled" workforce commit in the code artefact. If the skill level average of the workforce is less than "optimally skilled", the value of this variable increases proportionately. This constant has to be calibrated based on data collected over multiple projects with the development team.

Code ver productivity is a variable used to represent the amount of code that can be verified per time unit (e.g., day). As can be seen in figure 9 it depends on the headcount of the workforce allocated to carry out the verification activity, the number of code artefacts that have to be verified i.e. the number of modules that their code artefact has to be verified (determined by *Number of document being processed per activity*), skill level average of the verification team and *Maximum code ver rate per person per day*. *Maximum code ver rate per person per day* is the amount of code that "optimally skilled" verifiers can verify every day.

Code dev productivity is a variable used to represent the amount of code that can be developed (initially developed or reworked) per time unit (e.g., day). Its value depends on the value of *Maximum code dev rate per day* and the average skill level of the development team. As the average skill level of developers increases this productivity increases proportionately. *Maximum code dev rate per day* is the amount of code that "optimally skilled" developers develop every day. Its value is calculated differently for initial development and rework. In both cases it depends of the *Code learning status* and the headcount of the allocated developers. However, besides these variables, for initial

development its value depends on *Initial code dev rate per person per day* and if rework its value depends on the number of code faults detected in the code and the amount of effort required for the correction of the faults. *Initial code dev rate per person per day* specifies the amount of code that each "optimally skilled" developer develops every day.

Code dev effort and *Code ver effort* are level variables used for keeping track of the amount of effort spent for code development and code verification activities respectively. However, since the time step used for simulation is a day and it might happen that a developer is allocated to a task that takes less than a day, these variables are not accurate indications of the amount of effort that were actually spent on the activities. *Actual code rework effort* and *Actual initial code dev effort* are level variables used to address this issue.

4.2.1.4 Code Phase State Flow View

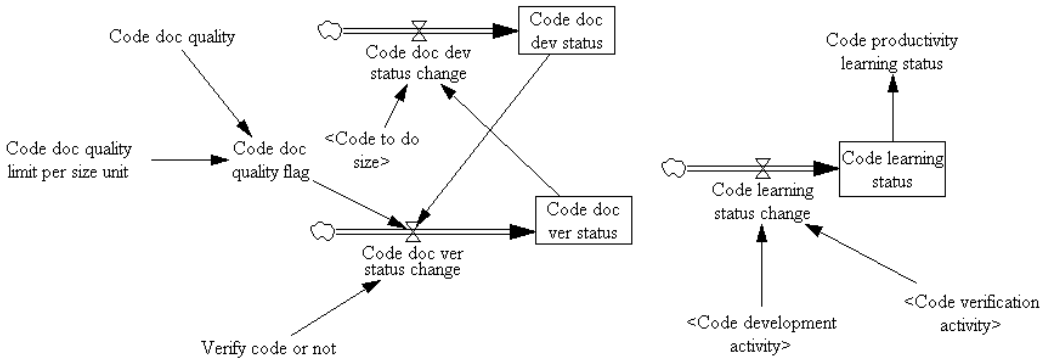
This view captures the specifics of how the states of different entities as explained in Section 3.1 change during the code phase of the development project. Figure 10 illustrates a simplified snapshot of this view with many of its variables hidden to improve readability and understandability of the graph.

Code doc dev status level variable is used to represent the state of the development activity (both initial development and rework). This level variable can have three different values. If its value is 0 it means that the development activity has not been started yet. As soon as some code artefact arrives for development (specified using the information link from *Code to do size* to *Code doc dev status change*), its value changes

to 1 meaning it is under process. Whenever code development is finished, value of this level variable is changed to 2.

Code doc ver status is used to represent the state of the verification activity. This level variable can have four different values. A value of 0 means that the activity has not been started yet. Whenever *Code doc dev status* becomes 2, i.e., code development is finished (specified using the information link from *Code doc dev status* to *Code doc ver status change*), its value changes to 1 meaning it is under process. Whenever the verification activity finishes its value changes depending on the number of detected code faults (specified using the *Code doc quality*) auxiliary variable.

Figure 10: Code Phase State Flow View



If *Code doc quality* (the number of code faults detected during the verification activity) is greater than or equal to a threshold specified using the *Code doc quality limit per size unit* constant, the value of the *Code doc quality flag* is set to 1 and the value of the *Code doc ver status* changes to 2, which means that the verification activity is finished but it has to be repeated once again due to bad quality. In this situation the value of *Code doc dev status* is set from 2 back to 1 (specified using the information link from the *Code dov dev status* to *Code doc dev status change*). If *Code doc quality* is smaller than the *Code doc*

quality limit per size unit constant, the value of *Code doc ver status* is set from 1 to 3, which means that it is complete and does not have to be repeated.

The *Verify code or not* constant is used to specify whether the code verification activity has to be carried out or not. If set to 1, the verification activity is carried out and the states of development/verification activities changes as described above. If set to 0, the verification activity is not carried out and the state of the verification activity maintains its initial value which is 0.

Code learning status level variable is essentially used to keep track of the number of times that the code artefact has been processed (specified by the information links from *Code development activity* and *Code verification activity* to *Code learning status change* rate variable). By processed, it is meant, developed, reworked or verified. Every time the code artefact is processed, value of this level is incremented by 1. *Code productivity learning status* specifies the effect of *Code learning status* on the productivity of the developers.

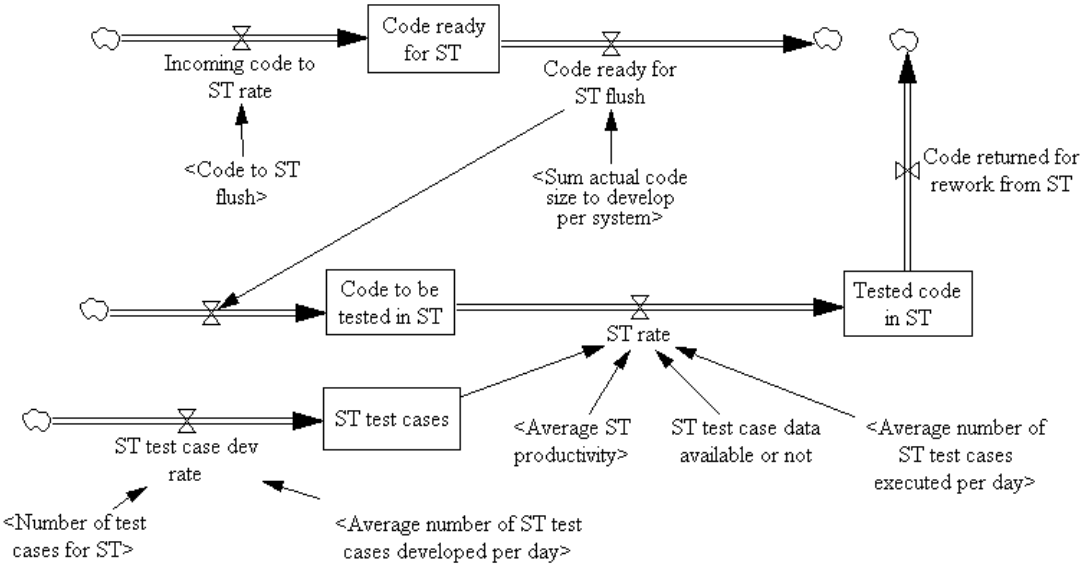
4.2.2 Validation Views

In this section, key underlying assumptions and mechanisms, levels, rates, and auxiliary variables implemented in the validation phases i.e. unit, integration and system testing of the process illustrated in Figure 6 are discussed in more detail. Since the second macro-pattern explained in Section 3.1 is applied to the product flow views of all the three validation phases (i.e. unit, integration and system testing), all the four views are quite similar for all validation phases. Therefore, here views related to only one of them, i.e., system testing is presented and explained.

4.2.2.1 System Testing Product Flow View

This view captures the specifics of how code artefacts are validated and sent back and forth between system testing validation and code phases of the development project. Figure 11 shows a simplified snapshot of the system testing product flow view with many of its auxiliary variables hidden to improve readability and understandability of the graph.

Figure 11: System Test Product Flow View



It is assumed that code artefacts of all modules of the system have to be ready for system testing before the system can go under system testing. *Incoming code to ST rate* is the rate variable used to represent the code artefacts that become ready for system testing and are sent to the system testing phase. *Code ready for ST* is the level variable used for keeping track of the code artefacts that are ready for system testing. It is emptied and all code artefacts are moved to *Code to be tested in ST* (Using the *Code ready for ST flush* rate variable) whenever code artefacts of all modules of the system arrive in the system

testing phase (represented by using the information link between *Sum actual code size to develop per system* to the *Code ready to ST flush* rate). When stored in the *Code to be tested in ST* level variable, system (i.e. code artefacts of the system) is waiting to be system tested.

ST test case data available or not is a flag used to indicate if empirical data about system test case development (e.g., productivity of test case development, number of test cases that need to be developed for system testing of the system, productivity of system test case execution, etc) is available for calibration or not. If available, this variable should be set to 1, otherwise 0.

If system testing test case calibration data is not available, system testing test case development and execution activities are combined together and considered as one system testing activity. In this situation system testing begins whenever workforce becomes available for system testing and the *Average ST productivity* and hence the *ST rate* becomes greater than zero. If system testing test case calibration data is available, system testing begins whenever workforce becomes available, and the number of developed test cases (represented using the *ST test cases* level variable) reaches the number of the required system test cases for the system (represented using the *Number of test cases for ST* variable). It is assumed that all test cases for the system have to be developed before system testing can begin. In this situation the *ST rate* i.e. the amount of code system tested everyday is determined by the value of *Average number of ST test cases executed per day*.

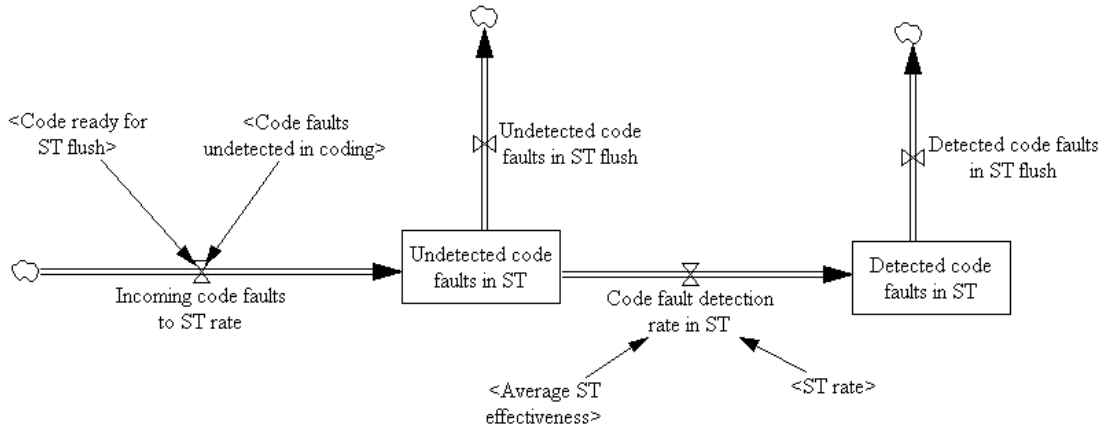
As the system testing is carried out, the amount of code artefacts that is tested is stored in the *Tested code in ST* level variable. After system testing is finished, the whole system is

sent back to the code phase for rework using the *Code returned for rework from ST* rate variable.

4.2.2.2 System Testing Defect Flow View

This view captures the specifics of how code faults are moved around i.e. propagated, detected and reported to and from the system testing phase. Figure 12 shows a simplified snapshot of this view with many of its auxiliary variables hidden to improve readability and understandability of the graph.

Figure 12: System Test Defect Flow View



Incoming code faults to ST rate is used to transition all the code faults existing in the system code artefacts (represented by the *Code faults undetected in coding*) to the system testing phase as the system becomes ready for system testing (represented using the *Code ready for ST flush*). Code faults propagated to the system testing phase are stored in the *Undetected code faults in ST* level variable where they wait for the system testing to begin. When system testing begins (represented using the information link from *ST rate* to the *Code fault detection rate in ST*), a portion of the defects in the system code artefacts are detected. This portion is determined using the *Average ST effectiveness*.

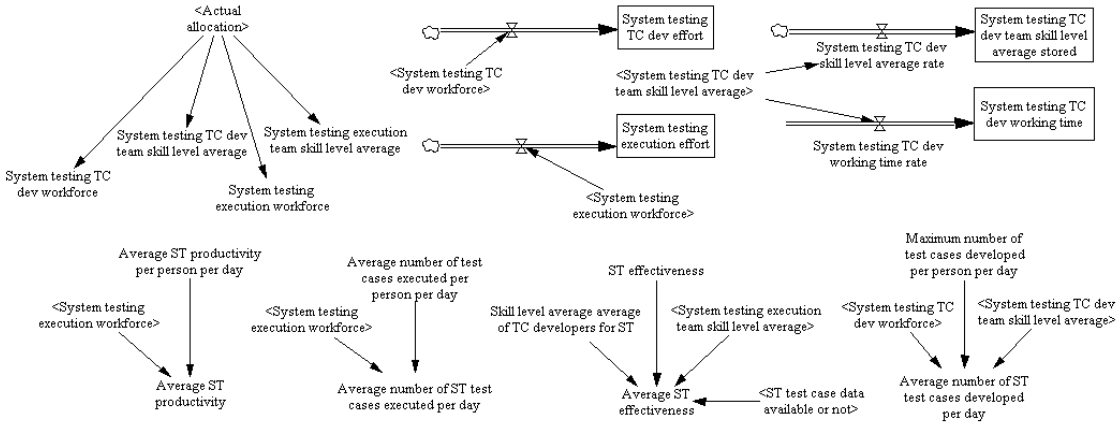
Detected code faults in ST level variable is used to keep track of the number of code faults detected during the system testing. Whenever system testing is finished, the detected code faults are reported to the code phase and the *Detected code faults in ST* level variable is emptied using the *Detected code faults in ST flush* rate variable. When system testing is finished, the *Undetected code faults in ST* level variable is also emptied using the *Undetected code faults in ST flush*. This is done so that this level is empty when system testing is carried out another time.

4.2.2.3 System testing Resource Flow View

This view captures the specifics of various attributes of different resources, i.e., developers and techniques/tools, used in the system test phase of the development project. Figure 13 illustrates a simplified snapshot of this view with some of its auxiliary variables hidden to improve readability and understandability of the graph.

Similar to code phase resource flow view, *Actual allocation* is used to specify the headcount and skill level average of the workforce allocated to system test case development and system test case execution. If system test case calibration data is not available, nobody is ever allocated to system test case development and the workforce allocated to system test case execution will carry out the system testing activity. *System testing TC dev effort* and *System testing execution effort* are level variables used to keep track the amount of effort spent on system test case development and system test case execution respectively.

Figure 13: System Test Resource Flow View



If system test case calibration data is available, it is assumed that the average skill level of the workforce that develops the system test cases effect the effectiveness of the system testing activity in detecting defects. *System testing TC dev team skill level average stored* is used to keep track of the skill level of different teams of workforce who work on system test case development. *System testing TC dev working time* level variable keeps track of the time that different teams work on system test case development. *Skill level average average of TC developers for ST* is the auxiliary variable used to calculate the average of average skill level of different teams who have worked on system test case development. The defect detection effectiveness of the system testing technique is changed proportionate to this average. If system test case calibration data is not available, the effectiveness of the system testing activity is changed proportionate to the average skill level of the system test execution team.

If system test case calibration data is available, the productivity of system testing, i.e., the amount of code artefact that is system tested per day is derived from the *Average ST productivity per person per day* constant and the *System testing execution workforce*

variable. The *Average ST productivity per person per day* is among the constant variables that have to be calibrated to empirical data. If system testing case calibration data is available, *Average number of ST test cases developed per day* is determined by the headcount of the workforce allocated to system test case development, the number of system test cases developed everyday by an "optimally skilled" developer (*Maximum number of test cases developed per person per day*) and the average skill level of the allocated team. It is assumed that average skill level of the system test execution team has no effect on their system test case execution productivity (represented by the *Average number of ST test cases executed per day*) since the test case execution activity is considered an automated procedure of running the test cases and reporting the results.

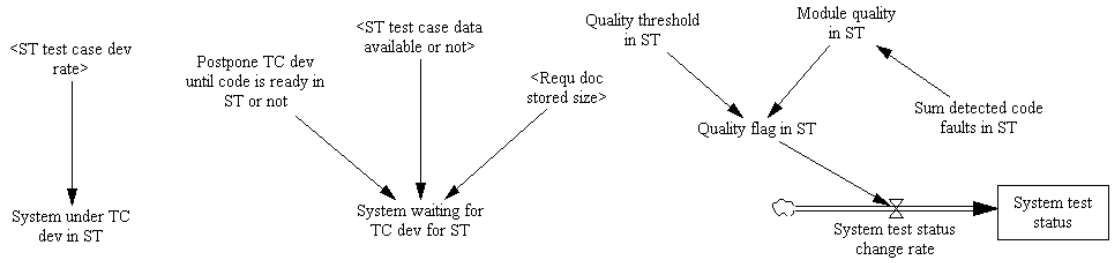
4.2.2.4 System testing State Flow View

This view captures the specifics of how the states of different entities as explained in Section 3.1 change during the system test phase of the development project. Figure 14 illustrates a simplified snapshot of this view with many auxiliary variables hidden to improve readability and understandability of the graph.

If system test case calibration data is available, *System under TC dev in ST* is used to represent the state of the test case development. If system test cases are being developed value of this auxiliary variable is set to 1, otherwise it is set to 0. *System waiting for TC dev for ST* is a flag used to specify if system test cases can be developed. *Postpone TC dev until code is ready in ST or not* is a constant used to represent the project's management decision about the time to develop the test cases. If value of this constant is set to 1, system test case development begins only when all the code artefacts of the

system are ready and if set to 0, system test case development can begin as soon as the requirements specification artefacts of the system are developed and verified.

Figure 14: System Test State Flow View



The *System test status* level variable is used to represent the state of the system testing activity. If system test case calibration data is available, it represents the state of the system test case execution activity. This level variable can have four different values. A value of 0 means that system testing has not been started yet. Whenever system testing begins, its value changes to 1 meaning it is under process. Whenever system testing finishes its value changes depending on the number of detected code faults (specified using the *Module Code doc quality*) auxiliary variable.

If *Code doc quality*, i.e., the number of code faults detected during system testing divided by the size of the system, is greater than a threshold specified using the *Quality threshold in ST* constant, the value of the *Quality flag in ST* is set to 1 and the value of the *System test status* changes to 2 which means that the system testing activity is finished but it has to be repeated once again due to bad quality. If *Code doc quality* is smaller than the *Quality threshold in ST* constant, the value of *System test status* is set from 1 to 3 meaning it is complete and it does not have to be repeated.

4.3 Subscripts

Subscription mechanism provided by Vensim® has been exploited in the implementation of GENSIM 2.0 to add to its reusability and to capture individual entities involved in the development project. The subscription mechanism in Vensim® is a feature that facilitates to have variables that calculate and hold multiple values for multiple entities simultaneously.

A subscript is an ordered set of entities in which each entity has a distinct identifier called subscript value. When a subscript is associated with a variable (using the subscript's name), the variable (variable's equation) is calculated for every entity in the subscript. The value of the variable for any individual entity is accessible using its subscript value.

For example, if a model user wants to capture the sizes of various modules of the system in one array, the model user can define a subscript named *module* and a variable called *module's size*. Assuming the system has three modules, the model user can define the *module* subscript values as *MOD1* for the first module, *MOD2* for the second module and *MOD3* for the third module. After associating *module's size* with *module*, *module's size[MOD1]* will specify the size of the first module, *module's size[MOD2]* will specify the size of the second module and *module's size[MOD3]* will specify the size of the third module.

The subscription mechanism adds much to the reusability of the model, because subscripted variables can be instantiated with different subscripts. In the example above, if the number of modules in the system changes from three to four modules, there is no need to change the *module's size* variable. The only necessary change is to modify the

module subscript and adding a fourth module with *MOD4* subscript value. Following is a list of subscripts used in GENSIM 2.0 along with their descriptions and current values.

Module is a subscript used to model individual modules of the system. Its subscript values are currently specified as *MOD1, MOD2, ..., MOD100* to represent a hypothetical system with 100 modules. However, it could easily be modified to represent different number of modules in the system.

Subsystem is a subscript used to model different subsystems within the system. Its subscript values are currently defined as *SUB1, SUB2, ..., SUB5* to model a hypothetical system with 5 subsystems. Like the *Module* subscript, it could be easily modified to model a system with a different number of subsystems.

It is assumed that every module in the system belongs to a distinct subsystem. This is achieved by specifying a mapping between the *Module* and the *Subsystem* subscripts.

Phase is a subscript used to capture individual phases of the development project. Its subscript values are currently specified as *RE, DE, CO, UT, IT* and *ST* representing requirements specification, design, code, unit test, integration test and system test respectively.

Origin is a subscript used to identify different origins that defects might have. Its subscript values are currently specified as *requ, design* and *code* representing requirements specification, design and code phases respectively. This subscript is generally associated with variables used to represent defects of software artefacts.

Factor is a subscript used to identify different aspects of software quality that defects have an effect on. By aspects of software quality, it is meant software quality characteristics as identified in the ISO 9126 [28] standard. This subscript enables

analyzing both functional and non-functional aspects of software quality. Its current values are defined as *RLB, USB* and *FUN* representing reliability, usability and functionality respectively. It is assumed that faults in the software artefacts could be characterized and differentiated by the quality aspect that they have the most significant effect on. Like the *Origin* subscript, it is generally associated with variables that capture defects of software artefacts and can be modified easily to enable evaluation of even more different aspects of quality.

Developer is a subscript used to capture individual employees available for the project. Its subscript values are currently specified as *DEV1, DEV2, ..., DEV40* to represent 40 employees available for the development project. However, it can be changed simply to model projects with different number of employees.

Activity is a subscript used to model single activities within the development project. Its subscript values are currently set as *RED, REV, DED, DEV, COD, COV, UTTC, UTV, ITTC, ITV, STTC* and *STV* to represent requirements specification development, requirements specification verification, design development, design verification, code development, code verification, unit test case development, unit test execution, integration test case development, integration test execution, system test case development and system test case execution.

4.4 Workforce Allocation Algorithm

The ability of Vensim® to work with external DLLs has been exploited in GENSIM 2.0 to extract organization-specific heuristics from the SD model and incorporating them into external DLL libraries where they can be changed easily without affecting the model structure. The algorithm that allocates developers to development, verification, and

validation activities is an example of an organization-specific heuristic that was implemented in a DLL library. The allocation function takes as input, the headcount and skill levels of the available workforce, workload of different activities and the minimum skill level required for the employees in order to be assigned to different activities.

Skill levels of the available workforce are represented by an $n \times m$ matrix S, as shown in Equation 2, in which n is the headcount of the available workforce, m is the number of activities which are carried out during the development life-cycle, and s_{ij} represents the skill level of the i^{th} employee in carrying out the j^{th} activity. As can be seen in the equation it is assumed that skill levels are given on a 0 to 1 continuous scale. If such accurate data does not exist within an organization, and the available data is on an ordinal scale, a simple mapping could resolve the issue (for details refer to Chapter 6).

Equation 2: Skill level Matrix S

$$S_{n\times m} = \begin{bmatrix} s_{11} & \cdots & s_{1m} \\ \vdots & \ddots & \vdots \\ s_{n1} & \cdots & s_{nm} \end{bmatrix}, s_{ij} \in [0,1]$$

Workloads of different activities are represented by an m-dimensional vector w, in which m is the number of activities and w_j represents the amount of work which is waiting to be done for the j^{th} activity. The value for w_j is determined by the number of artefacts, e.g., code modules which are waiting to be processed.

Minimum required skill levels of different activities are represented by an m-dimensional vector R, in which m is the number of activities and r_j specifies the minimum required skill level for the j^{th} activity.

To prepare the allocation of employees to tasks, using S and R, a new $n \times m$ matrix C is constructed, in which c_{ij} is set to 1, if $s_{ij} \geq r_j$, and set to 0, $s_{ij} < r_j$. The entry c_{ij} determines whether the i^{th} employee can be assigned to carry out the j^{th} activity having at least the activity's required skill level.

To each activity $j \in \{1, \ldots, k\}$ with $w_j > 0$, employees are assigned using the following algorithm:

- Step 1: assign to activity j all employees that can only carry out the j^{th} activity
- Step 2: assign to activity j a portion of the employees which can carry out only two activities including the j^{th} activity and one of the other activities for which $w_j > 0$
- Step 3: assign to activity j a portion of the employees which can carry out only three activities including the j^{th} activity and 2 of the other activities for which $w_j > 0$
- ⋮
- Step k: assign to activity j a portion of the employees which can carry out only k activities including the j^{th} activity and k-1 of the other activities for which $w_j > 0$

Each step t must be performed $\binom{k-1}{t-1}$ times to account for all the possible permutations.

The portion of the employees which will be assigned to the j^{th} activity in the t^{th} step for any of the possible permutations is determined using the formula shown in Equation 3. As the following calculations may result in floating point numbers, results are rounded if necessary.

Equation 3: Portion of developers assigned to activity *j* in step *t*

$$P_j = \frac{w_j}{w_j + \sum_{i \in I(j)} w_i} \quad with \ I(j) = \{ t-1 \ activities \ \ other \ \ than \ \ j \}$$

Chapter Five: Calibration of GENSIM 2.0

Model calibration refers to the adjustment of the simulation model calibration parameters until, for a certain input, the model's generated output matches a dataset observed in a real-world environment. Software process simulation model calibration can be done based on expert estimates or through parameter fitting based on historic data collected from organization-specific or public repositories. In cases when such data is not available, calibration can also be done using data published in the software engineering literature, which is in fact a mix of different sources. For the current version of GENSIM 2.0, data published in the literature was used. Calibration is an important step in the development of simulation models and is intended to ensure sound behaviour of the model. Calibration is also required to build confidence in the simulation results.

GENSIM 2.0 is designed to be reused, customized and applied to tackle emerging software development related problems of any kind. This chapter is not only intended to show the current calibration values of GENSIM 2.0, but also to elaborate more on its calibration parameters themselves which in turn allows for easy re-calibration of the model.

The calibration of SD (System Dynamics) models and the associated difficulties have been discussed in the literature for many years. The approach used for calibrating GENSIM 2.0 can be characterized as calibration "by hand". Calibration "by hand" is an iterative process in which the modeller "examines differences between simulated output and data, identifies possible reasons for those differences, adjusts model parameters in an effort to correct the discrepancy, and re-simulates the model, looping back to the first step." [29]. A discussion of the problems associated with this approach, i.e., limited

reliability of the calibration process which strongly relies on the expertise of the modeller, can be found in [29]. The alternative to calibration "by hand" is automated calibration. Automated calibration, while improving reliability, is not an easy endeavour either. A thorough discussion of the problems associated with automatic calibration of SD models, and ways to mitigate these problems, can be found in [30].

In [31], the authors argue that in order to truly realize the benefits of the use of software process simulation models, i.e., as virtual software engineering laboratories, process simulation has to be combined with empirical studies. They propose that this combination should be done in such a ways that firstly, empirical knowledge is used for development and calibration of simulation models and secondly, results from process simulation are used for supporting real experiments. They provide guidelines on how to achieve these objectives as well. The idea of combining simulation models with empirical studies is not limited to [31]. Pervious studies such as [32] had proposed the combination of process simulation models and empirical data in support of decision analysis in software development. In [32] the authors argue that since for decision-making, in practice, there is a need to apply the available empirical knowledge, expensive empirical work should be systematically extended with simulation to fill the gaps in the variable space of the decision-making context.

The approach taken for the purpose of this thesis is not different from the work discussed above from a methodological point of view. The only difference is that calibrating the model is explained for the entire set of model parameters which enables easier re-calibration and extended experimentation with the model as suggested in [31].

5.1 GENSIM 2.0 Calibration Parameters

Since, as discussed in chapter 4, corresponding sets of parameters (including calibration parameters) have been defined for modeling each development phase, Table 3 list and describe the calibration parameters used in modeling of the code development phase of GENSIM 2.0. For the same reason Table 4 list and describe the calibration parameters used in modeling the system test phase of GENSIM 2.0. Corresponding tables could be defined for other development or validation phases.

Table 3: Calibration parameters of the code development phase

	Parameter Name	Unit	Attribute of	View	Description
1	Code rework effort for code faults detected in CI	PD /Defect	Process	C-D	The amount of effort that should be spent for fixing a defect in the code artefacts if detected during code verification
2	Code rework effort for code faults detected in UT	PD /Defect	Process	C-D	The amount of effort that should be spent for fixing a defect in the code artefacts if detected during unit test
3	Code rework effort for code faults detected in IT	PD /Defect	Process	C-D	The amount of effort that should be spent for fixing a defect in the code artefacts if detected during integration test
4	Code rework effort for code faults detected in ST	PD /Defect	Process	C-D	The amount of effort that should be spent for fixing a defect in the code artefacts if detected during system test
5	Average design to code conversion factor	KLOC /Page	Product	C-P	The multiplier specifying how many kilo lines of code has to be developed per each page of the design artefacts.
6	Average # of UT test cases per code size unit	Test case /KLOC	Product	C-P	The number of test cases that has to be developed in order to unit test the code artefacts
7	Average design to code fault multiplier	N/A	Product	C-D	The multiplier specifying how many faults will be committed in the code artefacts because of an

					undetected design defect.
8	Maximum code ver. effectiveness	N/A	Resource	C-D	The effectiveness of the code verification technique with regards to defection of code defects if applied by an optimally skilled verifier
9	Maximum code ver. rate per person per day	KLOC /PD	Resource	C-P	The amount of code artefacts that can be verified in one day by an optimally skilled verifier.
10	Initial code dev. rate per person per day	KLOC /PD	Resource	C-R	The amount of code artefacts that can be developed (not reworked) in one day by one optimally skilled developer
11	Minimum code fault injection rate per size unit	Defect /KLOC	Resource	C-D	The number of defects that an optimally skilled developers in a size unit of the code artefacts.

CI= Code Inspection, UT= Unit Test, IT= Integration Test, ST= System Test, PD= Person-Day

Table 4: Calibration parameters of the system test phases

	Parameter Name	**Unit**	**Attribute of**	**View**	**Description**
1	Maximum ST effectiveness	N/A	Resource	S-D	The effectiveness of the system testing technique with regards to defection of code defects if applied by an optimally skilled tester
2	Average ST productivity per person per day	KLOC/ PD	Resource	S-P	The amount of code artefacts that can be system tested (including test case development) in one day by a tester. Since the test case development and execution activities are considered together in this parameter, It is assumed that the skill level of the testers does not have any effect on this rate.
3	Maximum # of ST test cases developed per person per day	Test case /PD	Resource	S-P	The number of system test cases that can be developed in one day by an optimally skilled tester.
4	Average # of ST test cases executed per person per day	Test case /PD	Resource	S-P	The number of system test cases that can be executed in one day by a tester. It is assumed that the skill level

					of the testers does not have any effect on the test case execution rate.

ST= System Test, PD= Person-Day

5.2 Sources of Calibration

Many different sources could potentially be used for calibration of software process simulation models, expert estimates are one of them. In [12] the authors discuss how expert knowledge and data from project data bases was used to develop and calibrate PSIM, a software process simulation model developed in order to show the usefulness of SD modeling in tackling software project management issues in a development department of Siemens. Despite being a good source in many situations, expert knowledge is considered to be subjective and is often complemented with real-world empirical data collected through measurement programs. Whenever simulation models are developed for specific purposes within specific organizations, this data could be gathered from different repositories within the organization. These repositories often contain data collected from many different projects carried out in the organization and are specific to its environment.

Whenever, expert knowledge or organization-specific data is not available or not applicable due to their relevance only in a specific context, online repositories (such as [33], [34] and [10]) which often contain cross-organizational data could be used to collect the necessary information for calibration of simulation models. The Software Information Repository [33], is maintained by an online community. Its members contribute and exchange information and data regarding process improvement activities around the world in order to build a knowledge base of this information which can be

used by any interested individual. The PROMISE Software Engineering Repository presented in [34] is a collection of datasets and tools is made publicly available to support researchers in building predictive software models. The Software-artefact Infrastructure Repository presented in [35] has been designed and constructed in support of controlled experimentation with software testing techniques. It contains many Java and C software systems, in multiple versions, along with their supporting artefacts such as test suites, fault data, scripts and manuals on how to experiment with the provided material. Results obtained from experimentation with the provided systems could be used to obtain estimates of different calibration parameters used in simulation models.

Another source of data that can be used for calibration of software process simulation models is the software engineering literature. The software engineering literature now contains many publications reporting data collected from different kinds of sources. Some of these publications such as [36] report data collected from an experiment carried out with groups of students or professionals. In [36] an experiment is carried out with one group of 42 advanced students and another group of 32 professionals to compare the effectiveness of three different testing strategies. Another group of these publications are the ones that report coarse-grained data gathered from real-world projects in industrial settings and mostly discuss the lessons learnt and the experiences. [37], is an example of these publications. In [37], the authors present high-level information regarding couples of projects in Motorola and discuss how CMM [38] helped them to improve the performance of their projects. The last group of publications comprises the results from surveys carried out on the existing literature itself. In [39] and [40] for example the authors have gathered and compared data reported in many other existing publications.

For the current version of GENSIM 2.0 the software engineering literature has been chosen as the source for calibration.

5.3 Current Parameter Calibration of GENSIM 2.0

In this section current values of GENSIM 2.0 calibration parameters together with how and from what source they were obtained are presented. The next subsection discusses the calibration of the development phases (requirements specification, design and code) and the in following subsection calibration of the validation phases are explained.

5.3.1 Calibration of the Development Phases

This subsection describes how and from which sources calibration parameters used in the simulation modeling of the development phases of GENSIM 2.0 were calibrated for its current version. To make the understanding of the overall material easier, the parameters are divided into different groups and then the calibration is presented for each group.

Initial development parameters: These parameters specify the productivity of the developers in developing software artefacts for the first time. For example, *Initial code dev. rate per person per day* specifies the speed with which the developers develop software code artefacts for the first time. Three parameters in the model belong to this category as shown in Table 5.

Table 5: Calibration values of the initial development parameters

Parameter Name	Unit	Value
Initial requ. dev. rate per person per day	Page/Person-Day	0.07
Initial design dev. rate per person per day	Page/Person-Day	0.829
Initial code dev. rate per person per day	KLOC/Person-Day	0.048

To calibrate these parameters, the COCOMO II [41] post architecture model was used. COCOMO II is a model that generates estimations of the cost, effort, and schedule of a

new software development activity across 4 life-cycle phases of software development, i.e., Plans and Requirements, Product Design, Programming and Integration and test. Additionally, within each life-cycle phase, it provides effort and time estimations for 8 different activities, i.e., Requirements Analysis, Product Design, Programming, Test Planning, Verification and Validation, Project Office, CM/QA and manuals.

In order to calculate the initial development parameters, specifications of a hypothetical system was inputted into the COCOMO II model and the estimations were generated. Following explains the details on how the Initial code dev. rate per person per day parameter was calculated:

The effort spent on the programming activity during the plans and requirements, product design and programming phases were summed up to obtain E. The schedule time spent on the programming activities of the three aforementioned phases were also summed up to obtain S.

The resultant effort (E) was divided by the resultant schedule time (S) to obtain the average number of developers allocated for the programming activity (D) during S.

It was assumed that S includes the time spent for the initial code development activity, rework due to faults detected in the code verification activity and rework due to defects detected in the unit test activity. Since, as explained in chapter 4, the effect of learning is incremented by 1 every time an artefact is processed (developed, reworked or verified), the learning effect is 2 right after it is verified completely for the first time and is 3 after it is completely reworked due the faults detected during verification. Hence, it is assumed as an approximation that the effect of learning equals an average of 2.5 during the period that it is reworked due to faults detected in code verification. As a result, rework due to

code faults detected in the code verification activity is 2.5 times faster than the initial development. For the same reason, rework due to defects detected in unit testing is assumed to be 3.5 times faster than the initial development.

With the above assumptions, solving Equation 4 for x will determine the overall schedule time spent on the initial code development activity (T).

Equation 4: Caculating the total initial code development duration

$$S = x + (1/2.5)x + (1/3.5)x$$

Having calculated T, the productivity of each of the developers in the initial code artefact development activity could be obtained using the formula shown in Equation 5 assuming the size of the code artefacts:

Equation 5: Calculating the initial code development productivity

$$p = \frac{Size\,of\ the\ \ code\ \ artifacts}{D \times T}$$

Corresponding calculations were applied to calculate the Initial design dev. rate per person per day and the Initial requ spec dev. rate per person per day parameters. In the hypothetical system specified for the current calibration of GENSIM 2.0, the size of the requirements specification artefact was assumed to be 50 pages. Considering the conversion factor between the requirements specification and the design artefacts, size of the overall design artefacts is calculated as 1550 pages by the simulation model. To calculate the size of the module code artefacts, their average size is calculated initially using the conversion factor between the design and code artefacts. After calculating the average module code size, in order to avoid having the same size for all the modules,

their average size is multiplied by a uniformly generated random number between 0.5 and 1.5. With these calculations, the size of the overall system is calculated as 307.15 KLOC.

Correction effort parameters: These parameters specify the amount of effort that needs to be spent for correcting the detected defects. The values of these parameters depend greatly on the time of the detection of the defect. For the code artefacts, to capture this difference, four distinct parameters where specified in GENSIM 2.0 as shown in the first four rows of Table 6. To calculate values of these four parameters two different sources were used ([39] and [42]). In [39], the author presents averages of many reported values in the literature for each of these four parameters as absolute numbers. In [42] the author presents these values as relative numbers. He claims that if the cost of fixing a defect if detected during the implementation is 1, then the cost of fixing it if detected during integration test is 2.5 and if detected during system test is 13. For the defects detected during code verification and unit test, values reported in [39] were used. For the defects detected in integration test and system test, we assumed that in [42], by defects detected during implementation, the author means, the defects detected during unit test. Hence, the value reported in [39] for the defects detected during unit test, was multiplied by 2.5 and 13 to calculate the parameters for integration and system test respectively.

For the requirements specification and design artefacts, since no data was found regarding the difference of the correction effort of the detected defects depending on the time of detection, the values reported in [39] were used to specify the amount of the effort required for correction of the defects of these artefacts regardless of the time of detection.

Table 6: Calibration values of the correction effort parameters

Parameter Name	Unit	Value
Code rework effort for code faults detected in CI	Person-Day/Defect	0.3387
Code rework effort for code faults detected in UT	Person-Day/Defect	0.4325
Code rework effort for code faults detected in IT	Person-Day/Defect	1.0815
Code rework effort for code faults detected in ST	Person-Day/Defect	5.6225
Design rework effort per fault	Person-Day/Defect	0.29
Requ. spec. rework effort per fault	Person-Day/Defect	0.125

Artefact Conversion Parameters: These parameters including *Average requ. spec. to design conversion factor* and *Average design to code conversion factor*, determine the relationship between the sizes of the artefacts developed in up-stream phases and the sizes of the artefacts developed in the down-stream phases. These parameters should be calibrated by collecting information from multiple projects for a certain context. Since no such data was found published in the software engineering literature, for the current calibration of GENSIM 2.0, these values where assumed hypothetically as shown in Table 7.

Table 7: Calibration values of the artefact conversion parameters

Parameter Name	Unit	Value
Average requ. spec. to design conversion factor	Page/Page	31
Average design to code conversion factor	KLOC/Page	0.2

Fault Conversion Parameters: These parameters determine the number of faults that an undetected defect in the artefacts of an up-stream phase causes in the artefacts of the down-stream phase. Similar to artefact conversion parameters, these parameters should be calibrated using information collected from multiple projects for a certain context. Since no such data was found published in the software engineering literature, for the

current calibration of GENSIM 2.0, these values where assumed hypothetically as shown in Table 8.

Table 8: Calibration values of the fault conversion parameters

Parameter Name	Unit	Value
Average requ. spec. to design fault multiplier	N/A	3
Average design to code fault multiplier	N/A	3

Verification Rate Parameters: These parameters determine the speed with which the verification activities are carried out. For calibration of these parameters the values presented in [39] were used as shown in Table 9.

Table 9: Calibration values of the verification rates

Parameter Name	Unit	Value
Maximum requ. spec. ver. rate per person per day	Page/Person-Day	8
Maximum design ver. rate per person per day	Page/Person-Day	30
Maximum code ver. rate per person per day	KLOC/Person-Day	0.6

Defect Injection Parameters: In GENSIM 2.0, defects in artefacts are caused by two different sources, firstly, the defects propagated from the up-stream phases and, secondly, the faults that the developers commit themselves. Defect injection parameters determine the rate with which the developers themselves inject defects in the artefacts. These parameters were calibrated using a Defect Containment Matrix [43]. A Defect Containment Matrix maps the phase in which a defect originated to the phase in which the defect was detected. For the current calibration of GENSIM 2.0, a hypothetical matrix, as illustrated in Table 10, was assumed and the injection rates were calculated according to this matrix. Rows represent the phases in which the defects are detected. Columns represent the origin of the detected defects. For example, it can be seen that a

total number of 2736 design defects were detected in the design phase, 1181 of which were originated in the requirements phase, i.e., were injected in the design artefacts due to undetected requirements specification defects.

Table 10: Defect Containment Matrix

	Stage Originated							
Stage Detected	**Requirements**	**Design**	**Code**	**UT**	**IT**	**ST**	**Field**	**Total**
Requirements	1515							1515
Design	1181	1555						2736
Code	402	912	2421					3735
Unit test	200	420	1525	37				2182
Integration test	191	223	370	7	1			792
System test	89	114	114	5	0	10		332
Field	5	8	23	0	0	0	0	36
Total	3583	3232	4453	49	1	10	0	11328

Using Table 10 we calculate the total number of defects that the developers have injected in different artefacts, i.e., requirements specification, design and code. In the following the calculation process is explained for the requirements specification artefact:

1. The number of requirements specifications defects that were detected in the design phase is calculated. From Table 10 it can be seen that 1181 design defects were detected in the design phase that originate in the requirement specification phase. Considering the fault multiplier between the requirements specification defects and the design defects (*Average requ. spec. to design fault multiplier*) which is assumed to be 3, it results that 394, i.e., 1181 divided by 3 requirements specification defects have been detected in the design phase.

2. The above calculation is done for the code defects that were found in the code phase and originated in the requirements specification phase. However, in this step, the applied fault multiplier is 9 because every requirements specification causes 3 design defects and every design defect causes 3 code defects. Hence, 45 (402 divided by 9) requirements specification defects were detected during the code phase.
3. The previous step with the same fault multiplier is carried out for the unit test, integration test, system test and the field phases.
4. The numbers of the requirements specification defects from all the previous steps are summed up together with the number of the requirements specification defects detected during the requirements specification phase itself to give us the total number of defects that the developers have injected in the requirements specification artefact.

A process corresponding to the above is followed for the design and code artefacts. The results of these calculations are shown in Table 11.

Table 11: Total number of defects injected in different artefacts

Artefact	Unit	Value
Requirements Specification	Defect	2007=1515+394+45+22+21+10
Design	Defect	2114=1555+304+140+74+38+3
Code	Defect	4453=2421+1525+370+114+23

Having the total number of defects that the developers have injected in different artefacts, the defect injection rates could be calculated by dividing the total number of defects injected in the artefacts by their size. The results of the calculations are shown in Table 12. The sizes of the artefacts were calculated as explained in the calibration of the initial development parameters.

Table 12: Calibration values of the defect injection parameters

Parameter Name	Unit	Value
Minimum requ. spec. fault injection rate per size unit	Defect/Page	40.14
Minimum design fault injection rate per size unit	Defect/Page	1.362
Minimum code fault injection rate per size unit	Defect/KLOC	14.52

Verification Effectiveness Parameters: These parameters specify the effectiveness of the artefact verification techniques with regards to defect detection. Effectiveness is expressed as the percentage of the artefact defects that are detected by the artefact verification technique. There exist many sources that can be used for calibration of these parameters in the software engineering literature, e.g., [39] and [40]. However, for the current calibration of GENSIM 2.0, we used the Defect Containment Matrix shown in Table 10 and the total number of defects injected in different artefacts shown in Table 11. In the following, the calibration is presented for the design verification technique.

To calculate the effectiveness of the design verification technique, we have to consider both the total number of defects that are injected in the design artefacts due to developer errors and the number of defects injected due to undetected requirements specification defects. Comparing Table 10 and Table 11, it can be seen that 492 (2007 minus 1515) requirements specifications defects propagate to the design phases. Considering the fault multiplier between requirements specification and design defects, it is concluded that 1477 (492 multiplied by 3) defects are injected in the design artefacts due to undetected requirements specification defects. Adding these 1477 defects to the 2114 design defects that are injected due to developer errors determines that 3591 defects have been injected in the design artefacts. According to Table 10, 2736 of these defects are detected in the design phase. Hence the effectiveness of the design verification technique is calculated as

0.76 (2736 divided 3591). A corresponding process is carried out for the requirements specification and code verification technique and the results are presented in Table 13.

Table 13: Calibration values of the verification effectiveness parameters

Parameter Name	Unit	Value
Maximum requ. spec. ver. effectiveness	N/A	0.75
Maximum design ver. effectiveness	N/A	0.76
Maximum code ver. effectiveness	N/A	0.53

Number of Test Cases Parameters: These parameters such as *Average # of UT test cases per code size unit* specify the number of test cases that is required to test the developed artefacts. Since no source for calibrating these parameters was found in the literature, no values have been assigned to these variables for the current calibration of GENSIM 2.0. Refer to chapter 4 for explanation on how the model works when test case data is not available.

5.3.2 Calibration of Validation Phases

This subsection describes how and from which sources calibration parameters used in the simulation modeling of the validation phases of GENSIM 2.0 were calibrated for its current version. To make the understanding of the overall material easier, the parameters are divided into different groups and then the calibration is presented for each group.

Validation Rate Parameters: These parameters specify the speed with which the testers test the artefacts. For calibration of these parameters the values represented in [39] as shown in Table 14 were used.

Table 14: Calibration values of the validation rate parameters

Parameter Name	Unit	Value
Average UT productivity per person per day	KLOC/Person-Day	0.3093
Average IT productivity per person per day	KLOC/Person-Day	0.1856
Average ST productivity per person per day	KLOC/Person-Day	0.1546

Validation Effectiveness Parameters: These parameters specify the effectiveness of the validation techniques with regards to defect detection. Similar to the verification effectiveness parameters, many sources are available in the software engineering literature to calibrate these parameters such as [39]. However, for the current calibration of GENSIM 2.0, similar to the verification effectiveness parameters, these parameters were calibrated using the Defect Containment Matrix shown in Table 10. Hence the effectiveness of each of these techniques was calculated by dividing the total number of code defects detected by the validation technique divided by the number of code defects propagated to the corresponding validation phase. The results of the calculations are shown in Table 15.

Table 15: Calibration values of the validation effectiveness parameters

Parameter Name	Unit	Value
Maximum UT effectiveness	N/A	0.66
Maximum IT effectiveness	N/A	0.69
Maximum ST effectiveness	N/A	0.93

Test case Development and Execution Rate Parameters: These parameters such as *Average # of UT test cases developed per person per day* specify the number of test cases that are developed or executed by one tester in one day. Since no source for calibrating these parameters was found in the literature, no values have been assigned to these

variables for the current calibration of GENSIM 2.0. Refer to chapter 4 for explanation on how the model works when test case data is not available.

Chapter Six: GENSIM 2.0 Application Scenarios

GENSIM 2.0 can be helpful in solving many different types of software process problems. In this chapter we discuss two of them as follows:

- Evaluating the overall effectiveness and efficiency of different combinations of development, verification, and validation techniques
- Analyzing the overall impact of changes to the workforce characteristics on project performance.

Initially, in sections 6.1 and 6.2, GENSIM 2.0 is used to find out the most effective and efficient combination of verification and validation techniques with regards to specific time, effort and quality goals. Secondly, in sections 6.3 and 6.4, it is used to find the most promising areas of investment in a project's workforce. It should be noted that the presented scenarios and analysis results are not intended to be evaluated objectively, but are intended to show examples of how GENSIM 2.0 can be applied and how its generated results can be analyzed.

6.1 Scenario 1: Choosing the best combination of V&V techniques with respect to project performance goals

One of the important features of GENSIM 2.0 is that its process structure can easily be modified. This scenario exploits this feature of GENSIM 2.0 to address the issue of finding the most suitable combination of verification and validation (V&V) activities considering time, effort and quality goals. It shows the impact of different combinations of V&V activities on project duration, product quality, and effort and how the model could assist decision makers in choosing with the best alternative. Verification activities include Requirements Inspections (RI), Design Inspections (DI) and Code Inspections

(CI). Validation activities include Unit Test (UT), Integration Test (IT), and System Test (ST). For each V&V activity there is exactly one technique with given efficiency (i.e., V&V rate) and effectiveness available. A V&V technique is either applied to all of the documents of the related type (e.g., requirements, design, and code documents) or it is not applied at all.

To clearly show how different calibration values can affect result analysis and how the model can assist decision-making in different development contexts, besides the original calibration of the model (Calibration B) this scenario was run using another calibration (Calibration A). The difference between these calibrations is shown in Table 16. In Calibration B, rework effort per detected defects is greater than in Calibration A for defects detected during integration and system testing.

Table 16: Difference between Calibration A and Calibration B

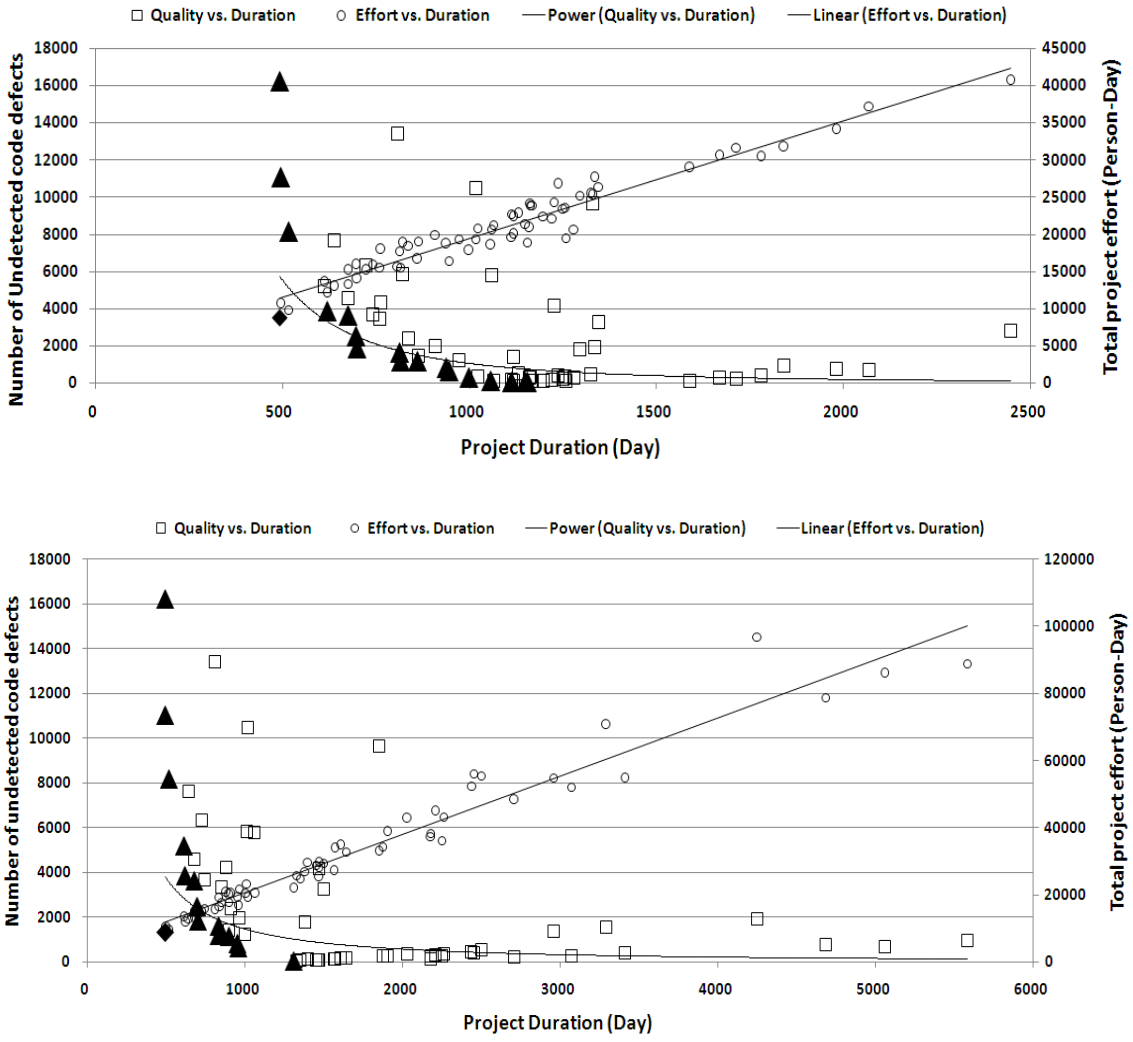
Calibration Parameter	Value	
	Calibration A	**Calibration B**
Code rework effort for code faults detected in IT	0.6775 PD/Def. [39]	1.0812 PD/Def. [[39],[42]]
Code rework effort for code faults detected in ST	1.0462 PD/Def. [39]	5.6225 PD/Def. [[39],[42]]

PD = Person-Day, Def. = Defect

Generating all the different V&V combinations and their respective simulation results was done automatically using the 'Sensitivity Analysis' feature provided in Vensim®. Figure 15 shows the simulation results for both calibrations of the model (Calibration A above / Calibration B below). Squares represent (Quality, Duration) result value pairs, where quality is measured as the total number of undetected code faults. Triangles

represent non-dominated (Quality, Duration) result values, i.e., simulation results to which no other simulation exists with both less undetected defects and less duration.

Figure 15: Quality vs. Duration and Effort vs. Duration (Scenario 1 – Calibrations A and B)



Obviously, for both calibrations, there exists a trade-off between Quality and Duration. Only looking at the non-dominated solutions, one can see that in order to achieve less undetected defects, more time is needed. In effect, if the goal was to see which combinations of V&V activities should be applied to achieve the target duration, in the

case that there are several eligible V&V combinations, a decision-maker could pick the non-dominated solution with the lowest number of undetected defects that is just within the project deadline.

Furthermore, a detailed analysis of all $2^6 = 64$ simulations using Table 17 which contains detailed results of the simulation runs per calibration reveals that the set of non-dominated solutions and their rankings differ for cases that involve IT and ST. For example, it turns out that with Calibration B, combination (RI, DI, CI, UT, -, ST) is better than combination (-, DI, -, -, IT, ST) with regards to duration, effort and quality. With Calibration A, combination (RI, DI, CI, UT, -, ST) is better than combination (-, DI, -, -, IT, ST) only with regards to effort and quality, but not with regards to project duration.

Table 17: Simulation results for Scenario 1

RI	DI	CI	UT	IT	ST	Duration [Day]		Effort [PD]		Quality [UD]	
						Cal. A	Cal. B	Cal. A	Cal. B	Cal. A	Cal. B
0	0	0	0	0	0	449	449	8640	8640	28508	28508
1	0	0	0	0	0	492	492	8836	8836	16241	16241
0	1	0	0	0	0	496	496	10639	10639	11054	11054
1	1	0	0	0	0	517	517	9825	9825	8171	8171
0	0	1	0	0	0	809	809	15701	15701	13422	13422
1	0	1	0	0	0	639	639	12986	12986	7656	7656
0	1	1	0	0	0	611	611	13602	13602	5218	5218
1	1	1	0	0	0	620	620	12085	12085	3850	3850
0	0	0	1	0	0	1017	1017	19370	19370	10498	10498
1	0	0	1	0	0	723	723	15193	15193	6343	6343
0	1	0	1	0	0	675	675	15195	15195	4597	4597
1	1	0	1	0	0	676	676	13342	13342	3633	3633
0	0	1	1	0	0	1060	1060	20655	20655	5775	5775
1	0	1	1	0	0	742	742	15952	15952	3663	3663
0	1	1	1	0	0	697	697	15908	15908	2499	2499
1	1	1	1	0	0	701	701	13989	13989	1845	1845
0	0	0	0	1	0	1329	1850	25282	33224	9641	9638
1	0	0	0	1	0	821	1008	18865	23339	5863	5840
0	1	0	0	1	0	764	877	17946	20936	4307	4242
1	1	0	0	1	0	761	850	15577	17765	3452	3369
0	0	1	0	1	0	1226	1469	24361	28104	4163	4163

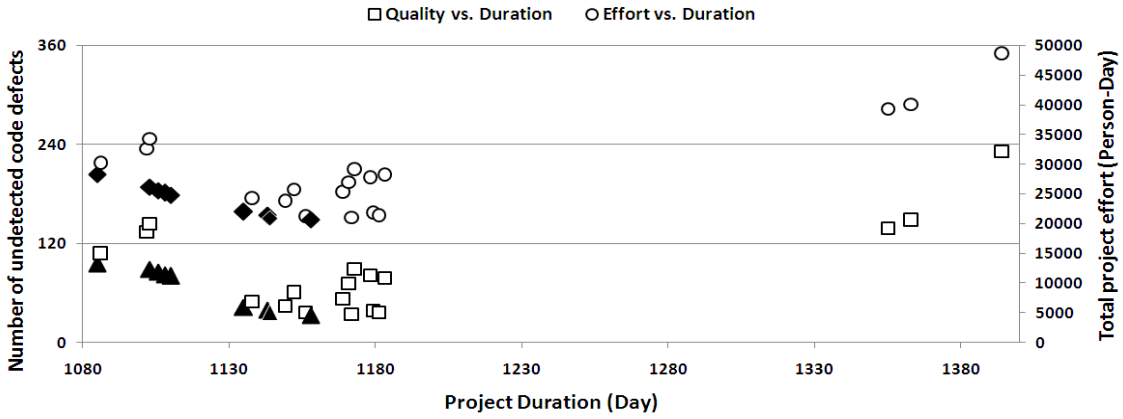
1	0	1	0	1	0	838	906	18490	20622	2376	2376
0	1	1	0	1	0	813	831	17792	19241	1621	1620
1	1	1	0	1	0	815	834	15532	16595	1197	1196
0	0	0	1	1	0	1346	1497	26432	29359	3277	3277
1	0	0	1	1	0	910	961	19984	21746	1993	1988
0	1	0	1	1	0	864	894	19050	20316	1455	1447
1	1	0	1	1	0	862	896	16666	17670	1154	1147
0	0	1	1	1	0	1295	1377	25261	26869	1791	1791
1	0	1	1	1	0	972	1000	19390	20490	1202	1224
0	1	1	1	1	0	939	951	18707	19457	824	832
1	1	1	1	1	0	946	958	16448	17005	626	629
0	0	0	0	0	1	2447	10400	40664	162002	2794	2793
1	0	0	0	0	1	1335	4246	27663	96784	1937	1934
0	1	0	0	0	1	1162	3288	23928	70974	1576	1572
1	1	0	0	0	1	1117	2956	20011	54788	1376	1370
0	0	1	0	0	1	1842	5581	31737	88857	942	942
1	0	1	0	0	1	1131	2498	22820	55405	539	539
0	1	1	0	0	1	1022	2026	20835	43041	368	368
1	1	1	0	0	1	1000	1873	17846	34232	272	272
0	0	0	1	0	1	1980	4682	34083	78759	760	760
1	0	0	1	0	1	1324	2435	25434	52430	469	469
0	1	0	1	0	1	1255	2259	23573	43136	347	347
1	1	0	1	0	1	1280	2250	20681	36140	280	279
0	0	1	1	0	1	1780	3408	30441	55019	405	405
1	0	1	1	0	1	1251	1904	23454	39044	257	257
0	1	1	1	0	1	1222	1641	22169	32806	176	176
1	1	1	1	0	1	1258	1564	19545	27398	130	130
0	0	0	0	1	1	2069	5057	37251	86209	697	697
1	0	0	0	1	1	1236	2450	26768	56075	433	431
0	1	0	0	1	1	1164	2205	24184	45164	324	319
1	1	0	0	1	1	1161	2174	20891	37339	264	258
0	0	1	0	1	1	1668	3071	30602	52063	292	292
1	0	1	0	1	1	1112	1604	22778	35053	167	167
0	1	1	0	1	1	1064	1395	21289	29630	114	114
1	1	1	0	1	1	1058	1349	18576	24735	84	84
0	0	0	1	1	1	1714	2704	31714	48584	232	232
1	0	0	1	1	1	1169	1571	23883	34080	142	142
0	1	0	1	1	1	1117	1468	22374	29790	104	104
1	1	0	1	1	1	1114	1465	19666	25545	83	83
0	0	1	1	1	1	1590	2177	28979	38223	126	126
1	0	1	1	1	1	1198	1449	22483	28818	84	86
0	1	1	1	1	1	1151	1326	21397	25697	58	58
1	1	1	1	1	1	1154	1308	18924	22160	44	44

PD = Person-Day, UD = Number of undetected defects in the code document

6.2 Scenario 2: Choosing the best combination of verification techniques with respect to project performance goals

This scenario uses only Calibration B as explained above. It shows the impact of different combinations of verification activities and techniques on project duration, product quality, and effort. This scenario assumes that all validation activities UT, IT, and ST are always performed, while verification activities (RI, DI, CI) can be performed or not. If a verification activity is performed, one of alternative techniques A or B can be applied. Compared to A-type verification techniques, B-type techniques are always 10% more effective (i.e., find 10% more of all defects contained in the related artefact) and 25% less efficient (i.e., 25% less amount of the related artefact can be verified per person-day).

Figure 16: Quality vs. Duration and Effort vs. Duration (Scenario 2 – Calibration B)



The simulation of all possible combinations generates $3^3 = 27$ different results (cf. Figure 16 and Table 18). Similar to what is shown in Figure 15, in Figure 16 the non-dominated solutions are marked by diamonds and triangles. The main difference to Scenario 1 is that in addition to the (Quality, Duration) trade-off there is a simultaneous (Effort, Duration) trade-off. For example, when having a closer look at Table 18 one notices that strictly

using B-type techniques in all performed verification activities will always result in less effort consumption and better quality than strictly using A-type techniques. With regards to duration, however, the picture is not so clear. While simulation results using patterns (B, 0, 0), (0, B, 0), (0, 0, B), (0, B, B), and (B, 0, B) indicate shorter duration than corresponding patterns using strictly A-type techniques, simulation results using patterns (B, B, 0) and (B, B, B) show longer durations than corresponding patterns (A, A, 0) and (A, A, A). When there is a mix of A-type and B-type verification techniques, the picture is even more complex and no general conclusions can be made, hence, the results have to be looked at in detail for each individual case.

Table 18: Simulation results for Scenario 2

Case	RI	DI	CI	RI-tech	DI-tech	CI-tech	Duration [Day]	Effort [PD]	Quality [UD]
1	0	1	0		B		1085	28401	96
2	0	1	0		A		1086	30272	108
3	1	0	0	B			1102	32830	135
4	1	0	0	A			1103	34448	145
5	1	1	0	A	A		1103	26216	89
6	1	1	0	B	A		1106	25637	86
7	1	1	0	A	B		1108	25220	82
8	1	1	0	B	B		1110	24892	81
9	1	1	1	A	A	A	1135	22032	43
10	0	1	1		B	A	1138	24297	50
11	1	1	1	B	A	A	1143	21516	40
12	1	1	1	B	B	A	1144	20926	36
13	0	1	1		B	B	1149	23888	45
14	0	1	1		A	A	1152	25916	61
15	1	1	1	A	B	A	1156	21253	38
16	1	1	1	B	B	B	1158	20728	33
17	0	1	1		A	B	1169	25412	54
18	1	0	1	B		B	1171	27094	72
19	1	1	1	A	B	B	1172	21153	35
20	1	0	1	A		A	1173	29187	90
21	1	0	1	B		A	1178	27836	82
22	1	1	1	A	A	B	1179	21865	40
23	1	1	1	B	A	B	1181	21434	38

24	1	0	1	A		B	1183	28302	79
25	0	0	1			B	1355	39446	139
26	0	0	1			A	1363	40287	149
27	0	0	0				1394	48683	233

PD = Person-Day, UD = Number of undetected defects in the code document

6.3 Scenario 3: Analyzing the effect of workforce headcount on project performance

The headcount of the workforce available for a project and their capabilities in carrying out different activities in the project have a significant impact on the project's performance. GENSIM 2.0 enables the project management to analyze this impact, taking into account all the mutual influences between the characteristics of the staffing profile, the sequence of activities, organizational policies for workforce allocation and other factors involved in the overall development process. The scenario presented in this section shows an example of the situations that GENSIM 2.0 could assist the management by providing estimates of the potential effects of the changes to a project's staffing profile.

To achieve increased reusability, in the implementation of GENSIM 2.0, organization-specific policies are extracted from the SD model and incorporated into external Dynamic Link Libraries (DLL) which allows for easy modification of these heuristics and algorithm. The workforce allocation is an example of such an algorithm. The current workforce allocation algorithm in GENSIM 2.0 which is also used for the purpose of the scenarios represented in this section is explained in section 4.4.

In this scenario, GENSIM 2.0 is used to evaluate the effect of doubling the headcount of a project's available workforce on its performance measures, i.e., effort, quality and duration. The analysis is performed on two extreme cases. In the first case, each activity is carried out by only one developer. In the second case, all activities can be carried out

by all available employees. Any other case between these extremes could be investigated similarly.

Case 1: In this case, the initial workforce consists of 6 employees and each activity can be carried out by only one of them. Hence, the staffing profile matrix could look like the example given in Equation 6. As can be seen, in this example each employee is capable of carrying out two consecutive activities. The simulation run with this staffing profile is referred to as the baseline run.

Equation 6: Initial Staffing profile matrix for Scenario 3 - Case 1

$$S_{6\times12} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

The effect of doubling the headcount of the workforce is analyzed in two different ways. Firstly, any of the activities can be carried out by only one of the employees and any of the employees can carry out only one activity. Therefore, the staffing profile matrix is defined as shown in Equation 7. The simulation run with this staffing profile is referred to as run A. Secondly, any of the activities can be carried out by two of the employees but each of these two can carry out two activities. Hence, the staffing profile matrix is specified as shown in Equation 8. The simulation run with this staffing profile is referred to as run B.

Equation 7: Staffing profile matrix for run A of Scenario 3 - Case 1

$$S_{12\times12} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 8: Staffing profile matrix for run B of Scenario 3 - Case 1

$$S_{12\times12} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Simulation results of run A, run B and the baseline run are shown in Table 19. It can be seen that because in run B each employee can carry out two activities and could be potentially allocated to any of them, run B yields a much greater improvement than run A with regards to the duration of the project. This could be explained by existing constraints inherent to the process structure. For example, requirements specification verification activity can only begin when the requirements specification development activity is

finished. Therefore, there is only a small overlap between the periods that each of these activities requires allocated workforce and that is the period when the verification is being carried out and the detected faults are being corrected meanwhile. As a result in run B, in most of the times when there is requirements specification development activity to be done, two developers are assigned to the activity and there is no competition between the development and the verification activities.

Table 19: Simulation results for Scenario 3 - Case 1

Run	Duration [Day]	Difference in Duration from baseline	Effort [PD]	Difference in effort from baseline	Quality [UD]	Difference in quality from baseline
Baseline	1088	0%	901	0%	2	0%
A	1080	-0.73%	1061	+17.75%	2	0%
B	610	-43.93%	1093	+21.30%	2	0%

PD = Person-Day, UD = Number of undetected defects in the code document

Quality remains the same in all the runs, because the skill levels of all the employees remain constant across different runs. The difference in the effort estimations is explained by the fact that the time step chosen for the simulation runs is one whole day. So, the employees are re-allocated to the activities on a daily basis. In cases that there is little work left to be done in any of the activities, the model still allocates workforce to that activity for the whole day which in turn causes the resulting effort estimations slightly different from the actual effort that has to be spent for that activity.

Case 2: In this case, the initial workforce consists of 6 developers and any activity can be carried out by any of the employees, i.e., each of them is capable of carrying out any of the activities. Hence, the staffing profile matrix is defined as illustrated in Equation 9. The simulation run with this staffing profile is referred to as the baseline run.

Equation 9: Initial Staffing profile matrix for Scenario 3 – Case 2

$$S_{6\times12} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The doubling effect is analyzed for a team of workforce with 12 developers with the same pattern of capabilities as the baseline run, i.e., each of the employees is capable of carrying out any of the activities. The simulation run with this staffing profile is referred to as run A. The simulation results of the two runs of this case are shown in Table 20.

Table 20: Simulation results for Scenario 3 – Case 2

Run	Duration [Day]	Difference in Duration from baseline	Effort [PD]	Difference in effort from baseline	Quality [UD]	Difference in quality from baseline
Baseline	280	0%	1005	0%	2	0%
A	154	-45%	1025	+2%	2	0%

PD = Person-Day, UD = Number of undetected defects in the code document

As shown in Table 20, the estimated duration of the project in run A is reduced by 45% percent. The reason why this effect is not estimated as 50% is that some work can be finished within one day whether 6 or 12 developers are allocated. As a result, since the simulation time step is one day, the duration of that particular one-day work remains equal for both runs. The difference in the effort estimates results from the same reasons as explained in Case 1.

Any case in between the above extreme cases, involving arbitrary settings of the staffing profile matrix, could be investigated in the same manner. For example, with the staffing

profile illustrated in Equation 10, duration of the project is estimated to be 232 days while the effort spent on the project is estimated to be 1135 person-days and the number of undetected defects in the code is estimated to be 2 defects.

Equation 10: Staffing profile matrix with arbitrary settings

$$S_{12\times12} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

6.4 Scenario 4: Analyzing the effect of workforce skill levels on project performance

This scenario shows the application of GENSIM 2.0 to analyze the effects of hiring better skilled workforce or training the current workforce on the project's performance and to figure out which type of skill sets is more worth the investment. GENSIM 2.0 assumes that for any of the employees, a skill level, a real number $s \in [0, 1]$, can be given for any of the activities of the project to specify his/her skill level in carrying out that activity. If providing the skill level with such accuracy is not possible and the employees' skill levels could only be specified on an ordinal scale a mapping from the ordinal scale onto [0,1] could resolve the issue. For example if the employees' skill levels is provided on an ordinal scale with five ordinals including excellent, good, medium, weak, and unable to do, we can map them onto [0,1] using a table as shown in Table 21.

Table 21: Example of mapping skill levels from ordinal to ratio scale

Value on ordinal scale	Value on ratio scale
Unable to do	0
Weak	0.25
Medium	0.5
Good	0.75
Excellent	1

The effect of the employees' skill levels on GENSIM 2.0 parameters is twofold. Whenever the skill level of an employee is increased, the speed with which he/she performs that activity is increased while his/her chances of making a mistake decreases. For example, if the skill level of testers is increased, the speed with which they test the artefacts increases while the effectiveness of the testing technique in defect detection increases. If the skill level of developers increases the speed with which they develop/rework the artefacts increases while the number of defects they inject in the artefact decreases. Because of the lack of reliable data on the magnitude of the effect of workforce skill levels on different parameters, it was assumed that all the parameters affected by the workforce skill levels increase/decrease proportionate to the average skill level of the employees i.e. if the average skill level of the system testers is 0.5, then the effectiveness of the system testing technique will drop from its reported optimal value by 50 percent.

The concrete question that this scenario answers is finding out the effect of hiring better skilled or training developers, verifiers and testers on project duration and effort and the quality of the final product. The question is analyzed for two example cases.

Case 1: In this case, the staffing profile matrix is specified as the run A of Case 2 of section 3.3, i.e., 12 employees that each of them could potentially carry out any of the

activities. The scenario includes four different simulation runs with differences in their inputs as illustrated in Table 22.

Table 22: Differences in inputs of the runs of Scenario 4 – Case 1

Run	Average skill level of developers	Average skill level of verifiers	Average skill level of testers
Baseline	0.5	0.5	0.5
A	0.75	0.5	0.5
B	0.5	0.75	0.5
C	0.5	0.5	0.75

Different output variables of the model could be used to analyze different effects. Table 23 includes the most important simulation results corresponding to the project's performance measures.

Table 23: Simulation results for Scenario 4 – Case 1

Run	Duration [Day]	Difference in Duration from baseline	Effort [PD]	Difference in effort from baseline	Quality [UD]	Difference in quality from baseline
Baseline	956	0%	4796	0%	502	0%
A	786	-17.78%	3924	-18.18%	418	-16.73%
B	500	-47.69%	2879	-39.97%	230	-52.3%
C	917	-4.07%	4645	-3.14%	158	-68.52%

PD = Person-Day, UD = Number of undetected defects in the code document

As shown in Table 23, if the main concern of the project management is the quality of the final product, improving the average skill level of the testers would result in more improvement compared to improving the average skill level of the verifiers or the developers. However if the effort or the duration of the project is considered as well, the third case yields the smallest improvement with regards to these factors. Thus, in order to decide on the group of workforce that is going to be invested in, priorities of the project

management have to be taken into account and trade-offs have to be analyzed. It is also worth to pint out why the third case shows more improvement in quality than the second case. This is due to the fact that the model assumes that testing techniques detect a certain percentage of the defects within the code regardless of the total number of defects in the code. The effect of the total number of defects in an artefact on the effectiveness of a related verification and validation technique was not considered in the model because no sufficient data was available.

Case 2: In this case, the workforce consists of 70 developers and each developer can potentially carry out only one activity. The number of developers that can carry out each of the activities is shown in Table 24. Besides its ability to generate estimates of the global effects of changes in a project's staffing profile, this scenario demonstrates that GENSIM 2.0 can easily handle staffing profiles of large development projects. Any of the simulation runs of this case take approximately 15 seconds which is very close to the elapsed time of the runs with the small cases.

Table 24: Workforce information for Scenario 4 – Case 2

Activity	Number of developers
Requirements specification development	6
Requirements specification verification	1
Design development	12
Design verification	3
Code development	23
Code verification	5
Unit test case development	5
Unit test	5
Integration test case development	5
Integration test	5
System test case development	10
System test	10

Four different simulation runs with differences in their input similar to those of Case 1 are analyzed. The results of the simulation runs are shown in Table 25.

Table 25: Simulation results for Scenario 4 – Case 2

Run	Duration [Day]	Difference in Duration from baseline	Effort [PD]	Difference in effort from baseline	Quality [UD]	Difference in quality from baseline
Baseline	545	0%	4872	0%	502	0%
A	446	-18.17%	3986	-18.19%	418	-16.67%
B	329	-39.63%	2984	-38.74%	230	-54.12%
C	524	-3.85%	4729	-2.92%	158	-68.60%

PD = Person-Day, UD = Number of undetected defects in the code document

It can be seen that, similar to the first case, if the major concern is quality of the final product, investing in the training the testers is the best choice. However, if duration and effort are important as well, investing in verifiers is the best alternative.

Chapter Seven: Conclusion and Future Work

7.1 Contributions

The contributions of the work presented in this thesis are as follows:

1. We identified a set of generic process structures (macro-patterns) including the development/verification pattern and the validation pattern. The former, captures the generic process structure of a software artefact development activity followed by an artefact verification activity. The latter, models the generic process structure of a software validation activity. The introduced patterns may be reused and composed in development of a wide range of software process simulation models and hence mitigate their respective cost issues.
2. To show the usefulness of the introduced macro-patterns, we developed a customizable software process simulator, i.e., GENSIM 2.0, by applying the identified macro-patterns to life-cycle phases of the well-known V-Model. GENSIM 2.0 captures the patterns from 4 different perspectives, flow of software artefacts, flow of defects, flow of resources and the changes in the states of the activities and artefacts. To provide a deeper insight into the specifics of the simulator and to enable it easy reuse and customization, we described all its parameters, their mutual influences, equations and other implementation details in this work.
3. We provided detailed description of all GENSIM 2.0 calibration parameters categorized in two groups, parameters used for calibrating the development/verification phases and the parameters used for calibration of the

validation phases. Currently calibration of GENSIM 2.0 is based on data available in the software engineering literature. Nevertheless, we discussed alternative sources that could potentially be used for the calibration of the model.

4. We described two example software process problems that GENSIM 2.0 can be applied to find a solution to. First we applied it to find the best combination of verification and validation activities with regards to specific time, quality and effort goals. Second we use GENSIM 2.0 to investigate the impact of changes to the project workforce on its overall performance. The presented application scenarios are not comprehensive but a subset of the variety of the situations that GENSIM 2.0 can be used in tackling software process issues.

7.2 Conclusion

Software process simulation plays a key role in improvement of software processes. Development of software process simulators has long suffered from cost issues. The work presented in this thesis is considered a major step towards more efficient development of these simulators. The macro-patterns introduced as part of this work are generic process structures that are used frequently in simulation modeling of software processes. Using the identified patterns, development of software process simulation models from scratch could be avoided.

GENSIM 2.0, developed as part of this work, by applying the macro-patterns, is a complex software process simulation model which has been made publicly available. Different to most SD software process simulation models, GENSIM 2.0 allows for detailed modeling of work products, activities, developers, techniques, tools, defects and

other entities from four different perspectives, i.e., product, defect, resource and state. Moreover, the possibility to use external DLL libraries gives the opportunity to extract organization-specific and potentially time-consuming algorithms from the SD model. Besides improving the simulator's performance, this feature allows for easy modification of such algorithms which are often hard-wired in the simulation model and cannot be changed easily. Comprehensive description of GENSIM 2.0 parameters and their calibration allows for easy re-calibration and further experimentation with the model.

7.3 Limitations and Future Work

Future work on GENSIM 2.0 will address some of its current limitations as follows:

- Future work regarding the calibration and validation of GENSIM 2.0 includes further investigation of the software engineering literature to find more sources that could potentially be used for re-calibration of GENSIM 2.0 in order to add to the reliability of its generated estimations.
- Future work regarding experimentation with GENSIM 2.0 will involve re-calibrating, reusing, customizing and applying it to deal with different kinds of software development process issues in real-world industrial development environments.
- The possibility to analyze the impact of the intensity level of V&V activities on project performance dimensions is also among the features that will be added to GENSIM 2.0 in future.
- GENSIM 2.0 has to be improved in order to capture additional characteristics of defects, e.g., severity. As a result, the simulator will be able to accurately capture defect prioritization before the correction activities and different defects could be

corrected at different points in time according to their urgencies. Furthermore, if such a defect prioritization scheme is used, a portion of defects may or may not be fixed at all.

- Currently, except requirement corrections incurred by corrections in code or design, GENSIM 2.0 does not allow any change in the requirements in the middle of a simulation run.
- Another present limitation of GENSIM 2.0 is that it assumes that the available workforce for the project is constant throughout the entire project. Future work on GENSIM 2.0 will address this issue and implement mechanisms for dealing with changeable workforce profiles.
- Currently it is not possible to represent incremental software development processes using GENSIM 2.0. Mechanisms will be added to the model that allow for concurrent execution of development cycles following the development process shown in Figure 6.
- Reusability of the macro-patterns has to be demonstrated empirically by collecting effort data from software process modeling projects that employ the patterns and comparing it with data collected from similar projects that do not apply them.

References

[1] M. Paulk, C. Weber, S. Garcia, M. B. Chrissis, and M. Bush, "Key practices of the capability maturity model," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 1993.

[2] H. Waeselynck and D. Pfahl, "System Dynamics Applied To The Modeling Of Software Projects," *Software Concepts and Tools,* vol. 15, no. 4, pp. 162-176, 1994.

[3] B. W. Bohem, C. Abts, A. W. Browm, S. Chulani, B. K. Clark, E. Horowitz, R. J. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*: Prentice-Hall, 2000.

[4] T. Abdel-Hamid and S. E. Madnick, *Software project dynamics: an integrated approach*: Prentice-Hall, Inc., 1991.

[5] S. Bandinelli, A. Fuggetta, L. Lavazza, M. Loi, and G. P. Picco, "Modeling and improving an industrial software process," *Software Engineering, IEEE Transactions on,* vol. 21, no. 5, pp. 440-454, 1995.

[6] V. Gruhn and A. Saalmann, "Software Process Validation Based on FUNSOFT Nets," in *Proceedings of the Second European Workshop on Software Process Technology*: Springer-Verlag, 1992.

[7] M. I. Kellner and G. A. Hansen, "Software process modeling: a case study," *System Sciences, 1989. Vol.II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, pp. 175-188 vol.2, 1989.

[8] P. Mi and W. Scacchi, "A knowledge-based environment for modeling and simulating software engineering processes," *Knowledge and Data Engineering, IEEE Transactions on,* vol. 2, no. 3, pp. 283-289, 1990.

[9] D. Pfahl, M. Klemm, and G. Ruhe, " A CBT module with integrated simulation component for software project management education and training," *Journal of Systems and Software,* vol. 59, no. 3, pp. 283-298, 2001.

[10] D. M. Raffo, U. Nayak, S. Setamanit, P. Sullivan, and W. Wakeland, "Using software process simulation to assess the impact of IV&V activities," *IEE Seminar Digests,* vol. 2004, no. 911, pp. 197-205, 2004.

[11] zz, "IEEE Standards Description: 12207.0-1996."

[12] D. Pfahl and K. Lebsanft, "Knowledge Acquisition and Process Guidance for Building System Dynamics Simulation Models. An Experience Report from Software Industry," *International Journal of Software Engineering and Knowledge Engineering,* vol. 10, no. 4, pp. 487-510, 2000.

[13] zz, "ExtendSim."

[14] zz, "Vensim."

[15] A. Wise, "Little-JIL 1.5 Language Report," Department of Computer Science, University of Massachusetts, Amherst UM-CS-2006-51, 2006.

[16] J. W. Forrester, *Industrial Dynamics*: M.I.T Press, 1961.

[17] P. Senge, *The fifth Discipline*. New York: Currency Doubleday, 1990.

[18] G. P. Richardson, *Feedback Thought in Social Science and Systems Theory*: University of Pennsylvania Press, 1990.
[19] C. Y. Lin, T. Abdel-Hamid, and J. S. Sherif, "Software Engineering Process Simulation Model (SEPS)," *Journal of Systems and Software,* vol. 38, pp. 263-277, 1997.
[20] R. J. Madachy, "A software project dynamics model for process cost, schedule and risk assessment," University of Southern California, 1994, p. 127.
[21] A. Powell and K. Mander, "Strategies for lifecycle concurrency and iteration: A system dynamics approach," *Journal of Systems and Software,* vol. 46, pp. 151-162, 1999.
[22] J. D. Tvedt, "An extensible model for evaluating the impact of process improvements on software development cycle time," Arizona State University, 1996, p. 386.
[23] R. Madachy, "Reusable Model Structures and Behaviors for Software Processes," in *Software Process Change*. vol. 3966/2006, Ed, Springer Berlin / Heidelberg, pp. 222-233.
[24] N. Angkasaputra and D. Pfahl, "Making Software Process Simulation Modeling Agile and Pattern-based," *ProSim 2004*, pp. 222-227, 2004.
[25] O. Armbrust, T. Berlage, T. Hanne, P. Lang, J. Munch, H. Neu, S. Nickel, I. Rus, A. Sarishvili, S. v. Stockum, and A. Wirsen, "Simulation-based software process modeling and evaluation," in *Handbook of Software Engineering & Knowledge Engineering*. vol. 3, Ed, 2005, pp. 333-364.
[26] M. Muller and D. Pfahl, "Simulation Methods," in *Guide to Advanced Empirical Software Engineering*, Ed, Springer London, 2008, pp. 117-152.
[27] D. Raffo, G. Spehar, and U. Nayak, "Generalized Process Simulation: What, Why and How?," *ProSim '03 Workshop*, 2003.
[28] zz, "The ISO 9126 Standard."
[29] J. M. Lyneis and A. L. Pugh, "Automated vs. "hand" calibration of system dynamics models: An experiment with a simple project model," *1996 International System Dynamics Conference*, pp. 317-320, 1996.
[30] R. Oliva, "Model calibration as a testing strategy for system dynamics models," *European Journal of Operational Research,* vol. 151, no. 3, pp. 552-568, 2003.
[31] J. Munch, D. Rombach, and I. Rus, "Creating an Advanced Software Engineering Laboratory by Combining Empirical Studies with Process Simulation," *Software Process Simulation Modeling (ProSim03)*, 2003.
[32] I. Rus, S. Biffl, and M. Halling, "Systematically combining process simulation and empirical data in support of decision analysis in software development," in *Proceedings of the 14th international conference on Software engineering and knowledge engineering* Ischia, Italy: ACM, 2002.
[33] zz, "Software Engineering Information Repository (SEIR)."
[34] zz, "PROMISE Software Engineering Repository."
[35] zz, "Software-artifact Infrastructure Repository."
[36] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *Software Engineering, IEEE Transactions on,* vol. SE-13, no. 12, pp. 1278-1296, 1987.

[37] M. Diaz and J. Sligo, "How software process improvement helped Motorola," *Software, IEEE,* vol. 14, no. 5, pp. 75-81, 1997.
[38] zz, "Capability Maturity Model for Software (CMM)."
[39] S. Wagner, "A literature survey of the quality economics of defect-detection techniques," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* Rio de Janeiro, Brazil: ACM, 2006.
[40] O. Laitenberger, "A Survey of Software Inspection Technologies," *Handbook on Software Engineering and Knowledge Engineering,* vol. 2, pp. 517-555, 2002.
[41] zz, "COCOMO II."
[42] L.-O. Damm, L. Lundberg, and C. Wohlin, "Faults-slip-through - a concept for measuring the efficiency of the test process," *Software Process: Improvement and Practice,* vol. 11, no. 1, pp. 47-59, 2006.
[43] A. A. Frost and M. J. Campo, "Advancing Defect Containment to Quantitative Defect Management," *The Journal of Defense Software Engineering,* vol. 20, no. 12, pp. 24-28, 2007.
[44] K. Khosrovian, D. Pfahl, and V. Garousi, "GENSIM 2.0: A Customizable Process Simulation Model for Software Process Evaluation," Schulich School of Engineering, University of Calgary SERG-2007-07, Simula TR 2008-01, 2007.

Appendix A

This appendix includes the GENSIM 2.0 equations as implemented in Vensim for the code development and the system test phases. For the complete set of equations and description of the variables refer to [44].

A.1. Model equation for the code development phase:

```
**********************************************************
        .Code Product Flow View
**********************************************************

Code to develop[module]=
        IF THEN ELSE(Design to CM[subsystem]>0:AND:Total code doc dev status per
subsystem[subsystem\
                ]=0, Random average code size in KLOC
        [module]/TIME STEP,0)
        ~       KLOC/Day
        ~               |
Code development activity[module]=
        IF THEN ELSE(Code doc dev status[module]=1,MIN(Code to do size[module]/TIME
STEP,Code dev productivity\
                [module]),0)
        ~       KLOC/Day
        ~               |
Verified code flush[module]=
        IF THEN ELSE(Code doc dev status[module]=1, Code doc verified[module]/TIME STEP,
0)
        ~       KLOC/Day
        ~               |
Code not to verify[module]=
        IF THEN ELSE(Code doc dev status[module]=2:AND:Code doc ver status[module]=3, Code
doc size\
                [module]/TIME STEP,IF THEN ELSE(Verify code or not
        =0,Code doc size[module]/TIME STEP,0))
        ~       KLOC/Day
        ~               |
Code to CM[module]=
        IF THEN ELSE(Code doc ready size[module]>Actual code size to develop per
module[module\
                ]*0.999, Code doc ready size[module
        ]/TIME STEP,0)
        ~       KLOC/Day
        ~               |

Random average code size in KLOC[module]=
        Code randomizing multipliers[module]*Average code size in KLOC[module]
        ~       KLOC
```

```
        ~       This variable specifies the size of different modules.
        |
Number of test cases for UT[module]=
        Average number of UT test cases per code size unit*Actual code size to develop per
module\
                [module]
        ~       Testcase
        ~               |
Code to IT flush[module]=
        IF THEN ELSE(Integration test or not=1:AND:Unit test or not=1:AND:Integration test
status\
                [module]<3:AND:Unit test status[module]>=3, Code to CM[module
        ],IF THEN ELSE(Integration test or not=1:AND:Unit test or not=0:AND:Integration
test status\
                [module]<3,Code to CM[module],0))
        ~       KLOC/Day
        ~       This rate is used to send a moduls's code document for integration
testing.
        |
Code to ST flush[module]=
        IF THEN ELSE(System test or not=1:AND:Integration test or not=1:AND:Integration
test status\
                [module]>=3:AND:System test status[module]<=2, Code to CM
        [module],IF    THEN    ELSE(System    test    or    not=1:AND:Integration    test    or
not=0:AND:System test status\
                [module]<=2,Code to CM[module],0))
        ~       KLOC/Day
        ~       This rate is used to send a moduls's code document for system testing.
        |
Code to UT flush[module]=
        IF  THEN  ELSE(Unit  test  status[module]<3:AND:Unit  test  or  not=1,  Code  to
CM[module], \
                0)
        ~       KLOC/Day
        ~       This rate is used to send a moduls's code document for unit testing.
        |
Code doc stored size[module]= INTEG (
        Code to CM[module]-Code to IT flush[module]-Code to ST flush[module]-Code to UT
flush\
                [module],
                0)
        ~       KLOC
        ~               |
Code verification activity[module]=
        IF    THEN    ELSE(Code    doc    ver    status[module]=1:OR:(Code    doc    ver
status[module]=2:AND:Code doc dev status\
                [module]=2),MIN(Code doc size
        [module]/TIME STEP, Code ver productivity),0)
        ~       KLOC/Day
        ~               |
Code to rework[module]=
        IF    THEN    ELSE(Code    doc    quality    ratio[module]>0,Code    verification
activity[module]*MIN\
                (1,Code  doc  quality  ratio[module])+Code  returned  for  rework  rate  from
UT[module]+Code returned for rework from IT\
```

```
		[module]+Code returned for rework from ST[module],Code verification
activity[module\
		]+Code returned for rework rate from UT[module]+Code returned for rework
from IT[module\
		]+Code returned for rework from ST[module])
	~	KLOC/Day
	~		|
Code not to rework[module]=
	MAX(Code verification activity[module]-Code to rework[module],0)
	~	KLOC/Day
	~		|
Number of modules per subsystem[SUB1]=
	ELMCOUNT(mod sub1) ~~|
Number of modules per subsystem[SUB2]=
	ELMCOUNT(mod sub2) ~~|
Number of modules per subsystem[SUB3]=
	ELMCOUNT(mod sub3) ~~|
Number of modules per subsystem[SUB4]=
	ELMCOUNT(mod sub4) ~~|
Number of modules per subsystem[SUB5]=
	ELMCOUNT(mod sub5)
	~	Dmnl
	~	This variable specifies the number of modules in every subsystem of the \
		product.
	|
Average code size in KLOC[module]=
	Average design to code conversion factor per subsystem[subsystem]*Average design
size in pages\
		[subsystem]/Number of modules per subsystem[subsystem]
	~	KLOC
	~	This variable specifies the average size of modules according to size of \
		their subsystem's design document and the number of modules in their \
		subsystem. It assumes that all modules within a subsystem correspond to \
		equal portions of their subsystem's design document size.
	|
Code doc verified[module]= INTEG (
	Code verification activity[module]-Verified code flush[module],
		0)
	~	KLOC
	~		|
Code doc quality ratio[module]=
	IF THEN ELSE(Code verification activity[module]>0:AND:Code doc quality limit per
size unit\
		>0, Sum code fault detection per module[module]/(Code doc quality limit
per size unit\
		*Code verification activity[module]), 0)
	~	Dmnl
	~		|
Actual code size to develop per module[module]= INTEG (
	Code to develop[module],
		0)
	~	KLOC
	~		|
```

```
Code doc ready size[module]= INTEG (
        Code not to rework[module]+Code not to verify[module]-Code to CM[module],
                0)
        ~       KLOC
        ~               |
Average design to code conversion factor per subsystem[subsystem]=
        0.2,0.14,0.18,0.25,0.2
        ~       KLOC/Page
        ~               |
Code to do size[module]= INTEG (
        Code to develop[module]-Code development activity[module]+Code to rework[module],
                0)
        ~       KLOC
        ~               |

****************************************************
```

.Code Defect Flow View

```
****************************************************

Code fault detection[origin,factor,module]=
        IF THEN ELSE( Actual code size to develop per module[module]>0:AND:Code
verification activity\
                [module]>0, MIN(Code faults undetected in coding
        [origin,factor,module]/TIME STEP, Average code ver
effectiveness[origin,factor]*Code verification activity\
                [module]*(Code faults undetected in coding
        [origin
        ,factor,module]*(Code doc size[module]+Code doc verified[module])/Actual code size
to develop per module\
                [module]+Code faults detected
        [origin,factor,module])/(Code doc size[module]+Code doc verified[module])),Code
fault detection rate in UT\
                [origin,factor
        ,module]+Code fault detection rate in IT[origin,factor,module]+Code fault
detection rate in ST\
                [origin,factor,module])
        ~       Defect/Day
        ~               |
Design to code fault propagation[origin,factor,subsystem]=
        IF THEN ELSE(Design to CM[subsystem]>0:AND:Total code doc dev status per
subsystem[subsystem\
                ]=0,(Design faults undetected
        [origin,factor,subsystem]*Average design to code fault multiplier[origin])/Number
of modules per subsystem\
                [subsystem]/TIME STEP,
        0)
        ~       Defect/Day
        ~               |
Code fault generation due to propagation[origin,factor,module]=
        IF THEN ELSE(Actual code size to develop per module[module]>0:AND:Design to code
faults waiting\
                [origin,factor,module]>0
        , MIN(Design to code faults waiting[origin,factor,module]/TIME STEP, (Design to
code faults waiting\
```

```
		[origin,factor,module]+Design to code faults propagated
	[origin,factor,module])*Code development activity[module]/Actual code size to
develop per module\
		[module]), 0)
	~	Defect/Day
	~		|
Corrected code faults flush[origin,factor,module]=
	IF THEN ELSE( Code doc dev status[module]=2 , Code faults
corrected[origin,factor,module\
		]/TIME STEP, 0)
	~	Defect/Day
	~		|
Detected code faults flush[origin,factor,module]=
	IF THEN ELSE( Code doc ver status[module]>1, Code faults
detected[origin,factor,module\
		]/TIME STEP, 0)
	~	Defect/Day
	~		|
Code ver effectiveness[requ,factor]=
	Code ver effectiveness constant ~~|
Code ver effectiveness[design,factor]=
	Code ver effectiveness constant ~~|
Code ver effectiveness[code,factor]=
	Code ver effectiveness constant
	~	Dmnl
	~		|
Code fault correction[origin,factor,module]=
	IF THEN ELSE(Code dev workforce per module[module]>0:AND:Code faults
pending[origin,\
		factor,module]>0,MIN(Code dev workforce per module[module]/(Code rework
effort per fault\
		[module]*9),Code faults pending[origin,factor,module]/TIME STEP),0)
	~	Defect/Day
	~		|
code fault generation[origin,factor,module]=
	Code fault generation due to propagation[origin,factor,module]+Code development
activity
	[module]*Average code fault injection per size unit
	[origin,factor]/MAX(1,Code learning status[module]^(Learning amplifier for code
fault injection\
		))
	~	Defect/Day
	~		|
Average code fault injection per size unit[origin,factor]=
	Minimum code fault injection per size unit[origin,factor]+(1-Code dev team skill
level average\
		)*Minimum code fault injection per size unit[origin,factor]
	~	Defect/KLOC
	~		|

Code faults pending[origin,factor,module]= INTEG (
	Code fault detection[origin,factor,module]-Code fault
correction[origin,factor,module\
		],
```

```
		0)
	~	Defect
	~		|
Code fault generation copy[origin,factor,module]=
	code fault generation[origin,factor,module]
	~	Defect/Day
	~		|
Design to code faults waiting[origin,factor,module]= INTEG (
	Design to code fault propagation[origin,factor,subsystem]-Code fault generation
due to propagation\
		[origin,factor,module],
		0)
	~	Defect
	~		|
Design to code faults propagated[origin,factor,module]= INTEG (
	Code fault generation due to propagation[origin,factor,module],
		0)
	~	Defect
	~		|
Sum code fault detection per module[module]=
	SUM(Code fault detection[origin!,factor!,module])
	~	Defect/Day
	~		|
Average design to code fault multiplier[origin]=
	3,3,0
	~	Dmnl
	~		|
Code faults detected[origin,factor,module]= INTEG (
	Code	fault	detection[origin,factor,module]-Detected	code	faults
flush[origin,factor,\
		module],
		0)
	~	Defect
	~		|
Actual code faults corrected[origin,factor,module]= INTEG (
	Code fault correction[origin,factor,module],
		0)
	~	Defect
	~		|
Actual code faults detected[origin,factor,module]= INTEG (
	Code fault detection[origin,factor,module],
		0)
	~	Defect
	~		|
Code faults corrected[origin,factor,module]= INTEG (
	Code	fault	correction[origin,factor,module]-Corrected	code	faults
flush[origin,factor\
		,module],
		0)
	~	Defect
	~		|
Code faults undetected in coding[origin,factor,module]= INTEG (
```

```
        Code      fault      generation      copy[origin,factor,module]-Code      fault
detection[origin,factor,\
                module],
                0)
        ~       Defect
        ~               |
Minimum code fault injection per size unit[requ,factor]=
        0,0,0 ~~|
Minimum code fault injection per size unit[design,factor]=
        0,0,0 ~~|
Minimum code fault injection per size unit[code,factor]=
        4.84,4.84,4.84
        ~       Defect/KLOC
        ~               |
Learning amplifier for code fault detection=
        2
        ~       Dmnl
        ~               |
Code faults generated[origin,factor,module]= INTEG (
        code fault generation[origin,factor,module],
                0)
        ~       Defect
        ~               |


********************************************************
```

.Code State Flow View

```
********************************************************

Code doc ver status change[module]=
        IF THEN ELSE((Code doc ver status[module]=0):AND:(Code doc dev
status[module]>1):AND:\
                Verify code or not=1,1,IF THEN ELSE
        ((Code doc ver status[module]=1):AND:(Code doc size[module]<=0):AND:(Code doc
quality flag\
                [module]>0):AND:Verify code or not=1,1,IF THEN ELSE((Code doc ver status
        [module]=2):AND:(Code doc size[module]>0):AND:(Code doc dev status[module]
        <>1):AND:Verify code or not=1,-1,IF THEN ELSE((Code doc ver
status[module]=1):AND:(Code doc size\
                [module]<=0):AND:(Code doc quality flag[module]<1):AND:Verify code or
not=1,2,IF THEN ELSE\
                (Code doc ver status[module]=0:AND:Verify code or not=0:AND:Code doc dev
status[module\
                ]>0,0,0)))))
        ~       Dmnl/Day
        ~               |
Code doc quality[module]=
        IF THEN ELSE(Code doc verified[module]>0, Sum code faults pending per
module[module]\
                /Code doc verified[module], 0)
        ~       Defect/KLOC
        ~               |
Code doc quality flag[module]=
        IF THEN ELSE(Code doc quality limit per size unit>0:AND:Code doc
quality[module]>Code doc quality limit per size unit\
                ,1,IF THEN ELSE(Code doc quality limit per size unit=0,0,0))
        ~       Dmnl
        ~               |
Code learning status change[module]=
        IF THEN ELSE(Actual code size to develop per module[module]>0, (Code development
activity\
```

```
               [module]+Code verification activity[module])/Actual code size to
develop per module\
               [module], 0)
        ~      Dmnl/Day
        ~             |
Code doc dev status change[module]=
        IF THEN ELSE((Code doc dev status[module]=0):AND:(Code to do size[module]>0),1,IF
THEN ELSE
        ((Code doc dev status[module]=1):AND:
        (Code to do size[module]<=0),1,IF THEN ELSE((Code doc dev
status[module]=2):AND:(Code to do size\
               [module]>0):AND:(Code doc ver status[module]<>1),-1,0))
        )
        ~      Dmnl/Day
        ~             |
Code doc dev status[module]= INTEG (
        Code doc dev status change[module],
               0)
        ~      Dmnl
        ~      status 0 : non_exist
               status 1: incomplete
               status 2: complete
        |
Code doc ver status[module]= INTEG (
        Code doc ver status change[module],
               0)
        ~      Dmnl
        ~      status 0 : non_exist
               status 1: incomplete
               status 2: complete_repeat
               status 3: complete_final
Code learning status[module]= INTEG (
        Code learning status change[module],
               0)
        ~      Dmnl
        ~             |
Code doc quality limit per size unit=
        0
        ~      Defect/KLOC
        ~             |

********************************************************
```

.Code Resource Flow View

```
********************************************************

Actual code dev effort per system=
        SUM(Actual code dev effort[module!])
        ~      Day*Person
        ~             |
Actual code dev effort rate[module]=
        IF THEN ELSE(Code learning status[module]<1,Code dev workforce per
module[module],0)
        ~      Person
        ~             |
Actual code dev effort[module]= INTEG (
        Actual code dev effort rate[module],
               0)
        ~      Day*Person
        ~             |
Code dev workforce per module[module]=
        IF THEN ELSE(Number of documents being processed per activity[COD]>0:AND:Code doc
dev status\
               [module]=1,Code dev workforce
        /Number of documents being processed per activity
        [COD],0)
        ~      Person
        ~             |
```

```
Actual code effort=
        Actual code dev effort per system+Sum actual code rework effort per system
        ~       Day*Person
        ~       This variable specifies the amount of actual effort spent on code \
                development/rework.
        |
Actual code rework effort[origin,factor,module]= INTEG (
        Actual code rework effort rate[origin,factor,module],
                0)
        ~       Day*Person
        ~               |
Actual code rework effort rate=A FUNCTION OF(Actual code rework effort rate,Code fault
correction\
                ,Code rework effort per fault) ~~|
Actual code rework effort rate[origin,factor,module]=
        Code fault correction[origin,factor,module]*Code rework effort per fault[module]c
        ~       Person
        ~               |
Code dev productivity[module]=
        Maximum code dev rate per day[module]*Code dev team skill level average
        ~       KLOC/Day
        ~               |
Sum actual code rework effort per system=
        SUM(Actual code rework effort[origin!,factor!,module!])
        ~       Day*Person
        ~               |
Initial code dev rate per person per day=
        0.048
        ~       KLOC/(Person*Day)
        ~               |
Code dev effort= INTEG (
        Code dev effort rate,
                0)
        ~       Day*Person
        ~               |
Code dev effort rate=
        Code dev workforce
        ~       Person
        ~               |
Code ver effort rate=
        Code ver workforce
        ~       Person
        ~               |
Code effort=
        Code dev effort+Code ver effort
        ~       Day*Person
        ~       This variable is used to show the total effort spent on the code document.
        |

Code ver effort= INTEG (
        Code ver effort rate,
                0)
        ~       Day*Person
        ~       This level variable is used to keep track of the effort spent of the code
\
                verification acitivity.
        |
Code ver team skill level average=
        Actual allocation[COV,SKLL]
        ~       Dmnl
        ~               |
Code ver workforce=
        Actual allocation[COV,NMBR]
        ~       Person
        ~               |
Code dev team skill level average=
        Actual allocation[COD,SKLL]
        ~       Dmnl
        ~               |
```

```
Average code ver effectiveness[origin,factor]=
        Code ver effectiveness[origin,factor]*Code ver team skill level average
        ~       Dmnl
        ~               |
Code ver productivity=
        IF THEN ELSE(Number of documents being processed per activity[COV]>0,(Code ver
workforce\
                /Number of documents being processed per activity
        [COV])*Maximum code ver rate per person per day*Code ver team skill level
average,0)
        ~       KLOC/Day
        ~               |
Maximum code ver rate per person per day=
        0.6
        ~       KLOC/(Day*Person)
        ~               |
Code ver effectiveness[requ,factor]=
        Code ver effectiveness constant ~~|
Code ver effectiveness[design,factor]=
        Code ver effectiveness constant ~~|
Code ver effectiveness[code,factor]=
        Code ver effectiveness constant
        ~       Dmnl
        ~               |
Average code fault injection per size unit[origin,factor]=
        Minimum code fault injection per size unit[origin,factor]+(1-Code dev team skill
level average\
                )*Minimum code fault injection per size unit[origin,factor]
        ~       Defect/KLOC
        ~               |
Minimum code fault injection per size unit[requ,factor]=
        0,0,0 ~~|
Minimum code fault injection per size unit[design,factor]=
        0,0,0 ~~|
Minimum code fault injection per size unit[code,factor]=
        4.84,4.84,4.84
        ~       Defect/KLOC
        ~               |
Code dev workforce=
        Actual allocation[COD,NMBR]
        ~       Person
        ~               |
```

A.2. Model equations for the System Test Phase

```
********************************************************
```

.ST Product Flow View

```
********************************************************

Code returned for rework from ST[module]=
        IF THEN ELSE(Tested code in ST[module]>0:AND:System test status[module]>1,Tested
code in ST\
                [module]/TIME STEP,0)
        ~       KLOC/Day
        ~               |
ST rate[module]=
        IF THEN ELSE(ST test case data available or not=1,IF THEN ELSE(Code to be tested
in ST\
                [module]>0:AND:ST test cases>Number of test cases for ST
        -0.0001:AND:Average number of ST test cases executed per day
        >0,MIN(Code to be tested in ST
        [module]/TIME STEP,Actual code size to develop per module[module]/(Number of test
cases for ST\
                /Average number of ST test cases executed per day
        )),0),IF THEN ELSE(Code to be tested in ST[module]>0:AND:Average ST
productivity>0,MIN\
```

```
                (Code to be tested in ST[module]/TIME STEP,
        Actual code size to develop per module[module]/(Sum actual code size to develop
per system\
                /Average ST productivity)),0))
        ~       KLOC/Day
        ~               |
Code ready for ST flush[module]=
        IF THEN ELSE(Sum code ready for ST<(Sum actual code size to develop per
system+0.1):AND:\
                Sum code ready for ST>(Sum actual code size to develop per system
        -0.1):AND:VMIN(Code doc dev status[module!])>0,Code ready for ST[module]/TIME
STEP,0\
                )
        ~       KLOC/Day
        ~               |
Sum code ready for ST per subsystem[subsystem]=
        CUSTOMSUMONED(Code ready for ST[MOD1],Subsystem's first module
number[subsystem],Number of modules per subsystem\
                [subsystem])
        ~       KLOC
        ~       This variable specifies the amount of a subsystem's code that is ready for
\
                system test.
        |
Sum code ready for ST=
        SUM(Sum code ready for ST per subsystem[subsystem!])+Sum code doc stored size per
system
        ~       KLOC
        ~               |
Code to be tested in ST[module]= INTEG (
        Code ready for ST flush[module]-ST rate[module],
                0)
        ~       KLOC
        ~               |
Tested code in ST[module]= INTEG (
        ST rate[module]-Code returned for rework from ST[module],
                0)
        ~       KLOC
        ~               |
Incoming code to ST rate[module]=
        Code to ST flush[module]
        ~       KLOC/Day
        ~               |
Code ready for ST[module]= INTEG (
        Incoming code to ST rate[module]-Code ready for ST flush[module],
                0)
        ~       KLOC
        ~               |

********************************************************
```

.ST Defect Flow View

```
********************************************************

Code fault detection rate in ST[origin,factor,module]=
        IF THEN ELSE(ST rate[module]>0, MIN(Undetected code faults in
ST[origin,factor,module\
                ]/TIME STEP, Average ST effectiveness[origin
        ,factor]
        *
        ST rate[module]*(Undetected code faults in ST[origin,factor,module]+Detected code
faults in ST\
                [origin,factor,module])/(
        Code to be tested in ST[module]+Tested code in ST[module])),0)
        ~       Defect/Day
        ~               |
Undetected code faults in ST flush[origin,factor,module]=
        IF THEN ELSE(System test status[module]>1,Undetected code faults in
ST[origin,factor\
```

```
                ,module]/TIME STEP,0)
        ~       Defect/Day
        ~               |
Incoming code faults to ST rate[origin,factor,module]=
        IF THEN ELSE(Code ready for ST flush[module]>0,Code faults undetected in
coding[origin\
                ,factor,module]/TIME STEP,0)
        ~       Defect/Day
        ~               |
Detected code faults in ST flush[origin,factor,module]=
        IF THEN ELSE(System test status[module]>1,Detected code faults in
ST[origin,factor,module\
                ]/TIME STEP,0)
        ~       Defect/Day
        ~               |
Undetected code faults in ST[origin,factor,module]= INTEG (
        Incoming code faults to ST rate[origin,factor,module]-Code fault detection rate in
ST\
                [origin,factor,module]-Undetected code faults in ST
flush[origin,factor,module],
                0)
        ~       Defect
        ~               |
Actual code faults detected in ST[origin,factor,module]= INTEG (
        Actual code faults detected in ST rate[origin,factor,module],
                0)
        ~       Defect
        ~               |
Actual code faults detected in ST rate[origin,factor,module]=
        Code fault detection rate in ST[origin,factor,module]
        ~       Defect/Day
        ~               |
ST effectiveness[requ,factor]=
        0.93,0.93,0.93 ~~|
ST effectiveness[design,factor]=
        0.93,0.93,0.93 ~~|
ST effectiveness[code,factor]=
        0.93,0.93,0.93
        ~       Dmnl
        ~               |
Detected code faults in ST[origin,factor,module]= INTEG (
        Code fault detection rate in ST[origin,factor,module]-Detected code faults in ST
flush\
                [origin,factor,module],
                0)
        ~       Defect
        ~               |

********************************************************
        .ST Status Flow View
********************************************************

Quality flag in ST[module]=
        IF THEN ELSE(Quality threshold in ST>0:AND:Module quality in ST[module]>Quality
threshold in ST\
                ,1,IF THEN ELSE(Quality threshold in ST=0,0,0))
        ~       Dmnl
        ~               |
System test status change rate[module]=
        IF THEN ELSE(ST rate[module]>0:AND:System test status[module]=0,1,IF THEN
ELSE(System test status\
                [module]=1:AND:ST rate[module]=0:AND:Quality flag in ST[module]=1,1,IF
THEN ELSE(System test status\
                [module]=1:AND:ST rate[module]=0:AND:Quality flag in ST[module]=0,2,IF
THEN ELSE(System test status\
                [module]=2:AND:ST rate[module]>0,-1,0))))
        ~       Dmnl/Day
```

```
        ~               |
Sum detected code faults in ST[module]=
        SUM(Detected code faults in ST[origin!,factor!,module])
        ~       Defect
        ~               |
Module quality in ST[module]=
        IF THEN ELSE(ST rate[module]=0:AND:Tested code in ST[module]>0,Sum detected code
faults in ST\
                [module]/Tested code in ST[module],0)
        ~       Defect/KLOC
        ~               |
System test status[module]= INTEG (
        System test status change rate[module],
                0)
        ~       Dmnl
        ~               |
Quality threshold in ST=
        0
        ~       Defect/KLOC
        ~               |

********************************************************
```

.ST Resource Flow View

```
********************************************************

Average ST productivity=
        System testing execution workforce*Average ST productivity per person per
day*System testing execution team skill level average
        ~       KLOC/Day
        ~               |
Skill level average average of TC developers for ST=
        IF THEN ELSE(ST TC dev done or not=1,System testing TC dev team skill level
average stored\
                /System testing TC dev working time*TIME STEP,0)
        ~       Dmnl
        ~               |
System testing TC dev skill level average rate=
        System testing TC dev team skill level average/TIME STEP
        ~       Dmnl/Day
        ~               |
System testing effort=
        System testing execution effort+System testing TC dev effort
        ~       Day*Person
        ~       This variable is used to show the total effort spent for system testing.
        |
System testing execution effort= INTEG (
        System testing execution workforce,
                0)
        ~       Day*Person
        ~               |
System testing TC dev effort= INTEG (
        System testing TC dev workforce,
                0)
        ~       Day*Person
        ~               |
Average ST effectiveness[origin,factor]=
        IF THEN ELSE(ST test case data available or not=1,ST
effectiveness[origin,factor]*Skill level average average of TC developers for ST
        ,System testing execution team skill level average*ST
effectiveness[origin,factor])
        ~       Dmnl
        ~               |
Average ST productivity per person per day=
        0.1546
        ~       KLOC/(Day*Person)
        ~               |
System testing TC dev workforce=
```

```
        Actual allocation[STTC,NMBR]
        ~       Person
        ~               |
System testing execution workforce=
        Actual allocation[STV,NMBR]
        ~       Person
        ~               |
System testing execution team skill level average=
        Actual allocation[STV,SKLL]
        ~       Dmnl
        ~               |
System testing TC dev team skill level average=
        Actual allocation[STTC,SKLL]
        ~       Dmnl
        ~               |
System testing TC dev team skill level average stored= INTEG (
        System testing TC dev skill level average rate,
                0)
        ~       Dmnl
        ~               |
System testing TC dev working time= INTEG (
        System testing TC dev working time rate,
                0)
        ~       Day
        ~               |
System testing TC dev working time rate=
        IF THEN ELSE(System testing TC dev team skill level average>0,1,0)
        ~       Dmnl
        ~               |
Average number of ST test cases developed per day=
        Maximum number of test cases developed per person per day*System testing TC dev
workforce\
                *System testing TC dev team skill level average
        ~       Testcase/Day
        ~               |
Maximum number of test cases developed per person per day=
        5
        ~       Testcase/(Day*Person)
        ~               |
Average number of ST test cases executed per day=
        Average number of test cases executed per person per day*System testing execution
workforce
        ~       Testcase/Day
        ~               |
Average number of test cases executed per person per day=
        20
        ~       Testcase/(Day*Person)
        ~               |
ST effectiveness[requ,factor]=
        0.93,0.93,0.93 ~~|
ST effectiveness[design,factor]=
        0.93,0.93,0.93 ~~|
ST effectiveness[code,factor]=
        0.93,0.93,0.93
        ~       Dmnl
        ~               |
```

Appendix B

This appendix includes the source code of the allocation function of GENSIM 2.0 in C++.

```
double GETALLOCATIONX(VECTOR_ARG *alloc,VECTOR_ARG *skills,VECTOR_ARG
*thresholds,VECTOR_ARG *workload,int num_activities,int num_devs)
{

        double required_skills_levels[12];
        double docs[12],temp_docs[12];
        int needy_activities[12];
        int permutation[12];
        int dev_cap_pattern[12];
        int num_needy_activities=0;
        double allocation[12];
        double sum_assigned_skill[12];
        int i,j,k,l,m,n,count=0,p,finalPerm,stringIndex,oneIndex;
        COMPREAL *allocated;
        double rval;

        double *dev_skills = alloca(num_devs*num_activities*sizeof(double));
        int *capabilities = alloca(num_devs*num_activities*sizeof(double));

        allocated = alloc->vals;

        for(i=0;i<num_devs*num_activities;++i)
                dev_skills[i]=skills->vals[i];

        //Initializing demands and skill level thresholds in local arrays
        for(i=0;i<num_activities;++i)
        {
                docs[i] = workload->vals[i];
                required_skills_levels[i] = thresholds->vals[i];
                allocation[i]=0;
                sum_assigned_skill[i]=0;
        }

        // Setting capabilities according to the required skill levels for
        // different activities
        for(j=0;j<num_activities;++j)
                for(i=0;i<num_devs;++i)
                {
                        if(required_skills_levels[j]>0.0 &&
dev_skills[i*num_activities+j]>required_skills_levels[j])
                                capabilities[i*num_activities+j]=1;
                        else if(required_skills_levels[j]==0.0 &&
dev_skills[i*num_activities+j]>0.0)
                                capabilities[i*num_activities+j]=1;
                        else
                                capabilities[i*num_activities+j]=0;
                }

        // Determining the number of activities which need personnel
        for(i=0;i<num_activities;++i)
                if(docs[i]>0)
                        num_needy_activities++;

        // Copying the docs array in a temp array to help with extracting
        // the index of needy activities
        for(i=0;i<num_activities;++i)
                temp_docs[i]=docs[i];

        // Determining the indexes of activities which need personnel.
        // The result is an array called needy_activities which has
```

```
	// the index of needy activities stored in it from its beginning
	// to the num_needy_activities
	for(i=0;i<num_activities;i++)
		for(j=0;j<num_activities;++j)
			if(temp_docs[j]>0)
			{
				needy_activities[i]=j;
				temp_docs[j]=0.0;
				break;
			}

	// The main for loop which is deciding on the share of the activity
	// from the developers that can carry out the task, considering other
	// needy activities.
	for(i=0;i<num_activities;++i)
	{
		if(docs[i]>0)
		{
			// Any developer that is going to be assigned to do the
			// activity must be capabale of it.
			dev_cap_pattern[i]=1;
			// Capabilities of the developer in activities which do
			// not need any personnel is not important. -1 in
			// dev_cap_pattern means that the developer's skill in
			// that activity is ignored.
			for(j=0;j<num_activities;++j)
				if(docs[j]==0)
					dev_cap_pattern[j]=-1;
			// Making capability patterns according to the following:
			// First developers that can just carry out this task
			// Second developers that can carry out 2 tasks including this task
			// Third developers that can carry out 3 tasks including this task
and ...

	//////////////////////////////////////////////////////////////////////
			// Assigninig developers which can only carry out this task. It is
separated
			// because the algorithm used to generate permutations cannot
generate the
			// permutation where all places are 0.
			for(j=0;j<num_needy_activities;j++)
				if(needy_activities[j]!=i)
					dev_cap_pattern[needy_activities[j]]=0;



	allocation[i]+=activity_share(i,dev_cap_pattern,capabilities,docs,num_activities,n
um_devs);

	sum_assigned_skill[i]+=activity_skill(i,dev_cap_pattern,capabilities,dev_skills,do
cs,num_activities,num_devs);

	//////////////////////////////////////////////////////////////////////
			// Assigninig developers that can carry out more than just this
task
			n = num_needy_activities-1;
			for(j=1;j<=n;++j)
			{
				p = j;
				for (k=0;k<n;k++)
					if (k < p)
						permutation[k] = 1;
					else permutation[k] = 0;
				// Look for the first generated capability pattern.
				for(l=0,m=0;l<num_needy_activities && m<n;l++,m++)

				{
					if(needy_activities[l]==i)
					{
```

```
                                        m--;
                                        continue;
                                }
                                else

        dev_cap_pattern[needy_activities[l]]=permutation[m];
                                }

        allocation[i]+=activity_share(i,dev_cap_pattern,capabilities,docs,num_activities,n
um_devs);

        sum_assigned_skill[i]+=activity_skill(i,dev_cap_pattern,capabilities,dev_skills,do
cs,num_activities,num_devs);

        ////////////////////////////////////////////////////////////////////
                                // Generating other permutations with the help of the first
one
                                for (finalPerm = 0; finalPerm < p; finalPerm++)
                                {
                                        for (oneIndex = finalPerm; oneIndex < p; oneIndex++)
                                                for (stringIndex = oneIndex; stringIndex <
n; stringIndex++)
                                                        if (permutation[stringIndex] == 0)
                                                        {
                                                                permutation[oneIndex] =
0;//new permutation
                                                                permutation[stringIndex] = 1;
                                                                count++;
                                                                // Form the dev_cap_pattern
according to the permutation
                                                                // and calling a proper
function to find out the task's
                                                                // share of the developers
with the specified pattern.

        for(l=0,m=0;l<num_needy_activities,m<n;l++,m++)
                                                                {

        if(needy_activities[l]==i)
                                                                        {
                                                                                m--;
                                                                                continue;
                                                                        }
                                                                        else

        dev_cap_pattern[needy_activities[l]]=permutation[m];
                                                                }

        allocation[i]+=activity_share(i,dev_cap_pattern,capabilities,docs,num_activities,n
um_devs);

        sum_assigned_skill[i]+=activity_skill(i,dev_cap_pattern,capabilities,dev_skills,do
cs,num_activities,num_devs);

        ////////////////////////////////////////////////////
                                                                permutation[oneIndex] =
1;//revert to old permutation.
                                                                permutation[stringIndex] = 0;
                                                        }
                                        permutation[finalPerm] = 0;//now set digit 1 to the
end of the number and repermutate
                                        permutation[n - finalPerm - 1] = 1;
                                }
                        }
                }
        }

        // Setting the returning vector
        for(i=0;i<num_activities;++i)
```

```
	{
		allocated[i*2]=allocation[i];
		if(allocation[i]>0)
			allocated[i*2+1]=sum_assigned_skill[i]/allocation[i];
		else
			allocated[i*2+1]=0;
	}

	rval = allocated[0];
	return rval;
}
////////////////////////////////////////////////////////////////
double activity_share(int activity,int *pattern,int *capabilities,double *workload,int
num_activities,int num_devs)
{
	double sum_workload=0.0;
	double raw_assignment[12];
	int num_matching_devs=0;
	int assigned_devs[12];
	int temp_assign[12];
	int dev_match,i,j,k,l,diff=0,max_index,min_index;
	int num_assigned_devs=0;
	int num_activity_to_adjust=0;
	int iterations=0;
	double temp_work[12];
	double sum;
	// Finding developers with capabilities matching the given pattern.
	for(i=0;i<num_devs;i++)
	{
		dev_match = 1;
		for(j=0;j<num_activities;++j)
			{
				if(pattern[j]==-1)
					continue;
				else if (pattern[j]!=capabilities[i*num_activities+j])
				{
					dev_match = 0;
					break;
				}
			}
		if(dev_match==1)
			num_matching_devs++;
	}
	////////////////////////////////////////////////////////////////
	// Summing up the amount of workload of the activities to which
	// developers will be assigned.
	for(i=0;i<num_activities;i++)
		if(pattern[i]==1)
			sum_workload+=workload[i];
	////////////////////////////////////////////////////////////////
	// Setting the initial assignment for different activities
	// If the raw assignment for a certain activity is below 1
	// then it is rounded up to 1 and if it is above 1 then it
	// is rounded down.
	for(i=0;i<num_activities;++i)
	{
		if(pattern[i]==1)
		{
			raw_assignment[i] =(workload[i]/sum_workload)*num_matching_devs;
			if((raw_assignment[i]<1) && (raw_assignment[i]>0))
				assigned_devs[i]=1;
			else
				assigned_devs[i]=(int)raw_assignment[i];
			num_assigned_devs = num_assigned_devs + assigned_devs[i];
		}
		else
			assigned_devs[i]=0;
	}
	////////////////////////////////////////////////////////////////////////
```

```
	// Adjusting the assigned numbers according to the available personnel
	// Since floating point numbers are rounded sometimes assigend personnel
	// are more than there are actually available and sometimes they are
	// less than available personnel.
	for(i=0;i<num_activities;++i)
		temp_assign[i]=assigned_devs[i];
	// If we have assigned less personnel than is actually available.
	if(num_assigned_devs<num_matching_devs)
	{
		diff = num_matching_devs-num_assigned_devs;
		while(diff>0)
		{
			max_index = find_max_indexx(temp_assign,num_activities);
			temp_assign[max_index]=-1;
			if(assigned_devs[max_index]>0)
			{
				assigned_devs[max_index]++;
				diff = diff - 1;
			}
		}

	}
	// If we have assigned more personnel that is actually available.
	if(num_assigned_devs>num_matching_devs)
	{
		for(i=0;i<num_activities;++i)
			if(assigned_devs[i]>0)
				num_activity_to_adjust++;

		for(i=0;i<num_activities;++i)
			if(assigned_devs[i]>0)
				temp_work[i]=workload[i];
			else
				temp_work[i]=0.0;
	}

	if(num_assigned_devs>num_matching_devs)
	{
		diff = num_assigned_devs-num_matching_devs;
		if(num_activity_to_adjust>0)
			iterations = diff/num_activity_to_adjust+1;

		for(i=0;i<(2*iterations);++i)
		{
			for(j=0;j<num_activities && diff>0;++j)
			{
				if(assigned_devs[j]>0)
				{
					min_index =
find_min_indexx(temp_assign,temp_work,num_activities);
					temp_assign[min_index]=0;
					temp_work[min_index]=1000;
					if(assigned_devs[min_index]>0)
					{
						assigned_devs[min_index]--;
						diff--;
					}
				}
			}
			for(k=0;k<num_activities;++k)
					temp_assign[k]=assigned_devs[k];


			for(l=0;l<num_activities;++l)
				if(assigned_devs[l]>0)
					temp_work[l]=workload[l];
		}
	}
```

```
	return assigned_devs[activity];
}
////////////////////////////////////////////////////////////
double activity_skill(int activity,int *pattern,int *capabilities,double *skills,double
*workload,int num_activities,int num_devs)
{
	double sum_workload=0.0;
	double raw_assignment[12];
	int num_matching_devs=0;
	int assigned_devs[12];
	int temp_assign[12];
	int *matching_devs;
	int *allocation;
	int dev_match,i,j,k,l,diff=0,max_index,min_index;
	int num_assigned_devs=0;
	int num_activity_to_adjust=0;
	int iterations=0,count;
	double temp_work[12];
	double sum;
	double sum_skill=0.0;

	matching_devs = alloca(num_devs*sizeof(int));
	allocation = alloca(num_devs*sizeof(int));

	// Finding developers with capabilities matching the given pattern.
	for(i=0;i<num_devs;i++)
	{
		dev_match = 1;
		for(j=0;j<num_activities;++j)
			{
				if(pattern[j]==-1)
					continue;
				else if (pattern[j]!=capabilities[i*num_activities+j])
				{
					dev_match = 0;
					break;
				}
		}
		if(dev_match==1)
		{
			num_matching_devs++;
			matching_devs[i]=1;
		}
		else if(dev_match==0)
			matching_devs[i]=0;
	}
	////////////////////////////////////////////////////////////////
	// Summing up the amount of workload of the activities to which
	// developers will be assigned.
	for(i=0;i<num_activities;i++)
		if(pattern[i]==1)
			sum_workload+=workload[i];
	////////////////////////////////////////////////////////////////
	// Setting the initial assignment for different activities
	// If the raw assignment for a certain activity is below 1
	// then it is rounded up to 1 and if it is above 1 then it
	// is rounded down.
	for(i=0;i<num_activities;++i)
	{
		if(pattern[i]==1)
		{
			raw_assignment[i] =(workload[i]/sum_workload)*num_matching_devs;
			if((raw_assignment[i]<1) && (raw_assignment[i]>0))
				assigned_devs[i]=1;
			else
				assigned_devs[i]=(int)raw_assignment[i];
			num_assigned_devs = num_assigned_devs + assigned_devs[i];
		}
		else
```

```
                assigned_devs[i]=0;
        }
        ////////////////////////////////////////////////////////////////////
        // Adjusting the assigned numbers according to the available personnel
        // Since floating point numbers are rounded sometimes assigend personnel
        // are more than there are actually available and sometimes they are
        // less than available personnel.
        for(i=0;i<num_activities;++i)
                temp_assign[i]=assigned_devs[i];
        // If we have assigned less personnel than is actually available.
        if(num_assigned_devs<num_matching_devs)
        {
                diff = num_matching_devs-num_assigned_devs;
                while(diff>0)
                {
                        max_index = find_max_indexx(temp_assign,num_activities);
                        temp_assign[max_index]=-1;
                        if(assigned_devs[max_index]>0)
                        {
                                assigned_devs[max_index]++;
                                diff = diff - 1;
                        }
                }

        }
        // If we have assigned more personnel that is actually available.
        if(num_assigned_devs>num_matching_devs)
        {
                for(i=0;i<num_activities;++i)
                        if(assigned_devs[i]>0)
                                num_activity_to_adjust++;

                for(i=0;i<num_activities;++i)
                        if(assigned_devs[i]>0)
                                temp_work[i]=workload[i];
                        else
                                temp_work[i]=0.0;
        }

        if(num_assigned_devs>num_matching_devs)
        {
                diff = num_assigned_devs-num_matching_devs;
                if(num_activity_to_adjust>0)
                        iterations = diff/num_activity_to_adjust+1;

                for(i=0;i<(2*iterations);++i)
                {
                        for(j=0;j<num_activities && diff>0;++j)
                        {
                                if(assigned_devs[j]>0)
                                {
                                        min_index =
find_min_indexx(temp_assign,temp_work,num_activities);
                                        temp_assign[min_index]=0;
                                        temp_work[min_index]=1000;
                                        if(assigned_devs[min_index]>0)
                                        {
                                                assigned_devs[min_index]--;
                                                diff--;
                                        }
                                }
                        }
                        for(k=0;k<num_activities;++k)
                                        temp_assign[k]=assigned_devs[k];


                        for(l=0;l<num_activities;++l)
                                if(assigned_devs[l]>0)
                                        temp_work[l]=workload[l];
```

```
                }
        }

        for(i=0,j=0;i<num_activities;++i)
        {
                if(assigned_devs[i]>0)
                {
                        count=assigned_devs[i];
                        while(count>0)
                        {
                                if(matching_devs[j]==1)
                                {
                                        count--;
                                        allocation[j]=i;
                                        j++;
                                }
                                else
                                        j++;
                        }
                }
        }

        for(i=0;i<num_devs;++i)
                if(allocation[i]==activity)
                        sum_skill+=skills[i*num_activities+activity];

        if(assigned_devs[activity]>0)
                return sum_skill;//assigned_devs[activity];
        else
                return 0;
}
```