University of Calgary, Technical Report, SERG-2007-07 Updated: June 19, 2008

Simula Research Laboratory, Technical Report, Simula TR 2008-01

GENSIM 2.0: A Customizable Process Simulation Model for Software Process Evaluation

Keyvan Khosrovian[1], Dietmar Pfahl[1, 2, 3], Vahid Garousi[1]

[1]Schulich School of Engineering, University of Calgary, Canada
[2]Simula Research Laboratory, Lysaker, Norway,
[3]Department of Informatics, University of Oslo, Norway
{kkhosrov, dpfahl, vgarousi}@ucalgary.ca

Abstract Software process analysis and improvement relies heavily on empirical research. Empirical research requires measurement, experimentation, and modeling. However, whatever evidence is gained via empirical research is strongly context dependent. Thus, it is hard to combine results and capitalize upon them in order to improve software development processes in evolving development environments. The process simulation model GENSIM 2.0 addresses the challenge mentioned above. Compared to existing process simulation models in the literature, the novelty of GENSIM 2.0 is twofold: (1) Model structure is customizable to organization-specific 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 ex-pert knowledge. This is achieved by making the internal model structures explicit and by providing guidance on how to calibrate model parameters. This technical report explains the overall structure of GENSIM 2.0, its internal mechanisms, and its parameters.

Keywords Software process simulation, System dynamics, Customizable process simulation model

TABLE OF CONTENTS

1 INTRODUCTION

Software industry has always been confronted with problems of overdue deadlines and cost overruns as well as with poor product quality delivered by software development organizations. Meanwhile, increasing demand for 'better, faster, cheaper' along with increasing complexity of software systems have urged software developers to bring discipline to the development of software systems and to improve its performance [1]. Two major factors affect software project performance [2]. Firstly there are technological issues, e.g., languages, tools, hardware, etc. Despite of the considerable improvements achieved in these areas, it has been seen that the extent to which these factors could impact project performance is limited. Secondly there are the managerial issues, e.g., planning, resource allocations, workforce training, etc 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 development, transforming the art of software development into an engineering discipline and hence improving overall performance of software development activities. Engineering disciplines on the other hand, require provision of evidence on the efficiency and effectiveness of tools and techniques in varying application contexts. In the software engineering domain, 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 organizational aspects such as process organization, 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 project.

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 are a means for assessing 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 application context, for example, the typical defect detection effectiveness of an inspection or test technique applied to a specific type of development artifact, 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 development project performance, i.e., total project duration, total project effort consumption (or cost), and quality of the end product delivered to the customers while considering all other entities involved in the development process and their mutual influences . Typically, global efficiency and effectiveness are evaluated through case studies.

Controlled experiments and case studies are expensive in terms of effort and time consumption. Therefore it is not possible to experiment with all alternatives in real projects. Nevertheless, support for deciding which experiments and case studies are more worthwhile to spend effort and time on would be helpful. 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 to what extent 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 overall project performance (total duration, total effort consumption, end product quality) 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 hypothetical improvements are worthwhile to be empirically investigated within various contexts.

In order to assist decision makers in situations described above and help minimize the drawbacks of the current decision making approaches, one can provide experts with a software process simulation system that generates estimates of the impact of local process changes on overall project performance. For example, if derived from case studies or laboratory experiments with the tool or technique, or from what the vendors of the tool claim, the defect detection effectiveness of unit testing technique *A* is assumed to be locally 10% better than that of unit testing technique *B* in a given context, through simulation we may find out that using technique *B* instead of technique *B* yields an overall positive impact of 2% on 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 a specific context).

With the help of the simulation models, even more complex situations could be investigated as well. 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 on more critical 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 to gain a better understanding of the 'big picture' of the development project.

Another example of the complex situations where the 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, i.e., time, cost and quality goals. Obviously, in general, hiring a large workforce dedicated to carrying out any of the activities within the development process will result in better project performance regarding time, but in a specific context, the management might not be able to do so due to specific budget and resource constraints. In such a situation, the simulation model could help the 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 process 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 more highly skilled developers improve different project performance dimensions. 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 project 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 skill level of different groups of workforce on the overall project performance by taking into account all specifics of the entire development project and assess whether and to what extent investments to increase workforce skills actually pay off.

A well-known issue related to process simulators is the question of their high development and maintenance costs. This report offers a solution to this issue. A core element of the proposed solution is the simulation framework GENSIM 2.0 (GENeric SIMulator, Version 2.0), which is a substantially enhanced version of an older research prototype (GENSIM [3]). Inspired by the idea of frameworks in software engineering, GENSIM 2.0 consists of a small set of generic reusable components which can be ideally put together 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 product/process structure, quality and resource specific attributes of these 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 artifact development activity and related verification activity (inspection) carried out on the developed artifact, and three validation (testing) activities, consisting of unit, integration and system test.

This technical report is structured as follows: Sections 1 and 2 motivate the research conducted and provide related background information about the System Dynamics (SD) simulation modeling technique. Section 3 discusses related work and how the research presented in this paper differs from existing research. Section 4 presents an overview of the structure of GENSIM 2.0 and its reusable process structure components. Section 5 describes details of GENSIM 2.0, more specifically how its reusable and easily adaptable model structure reflects generic software process structures, what model parameters are used for input, output, and model calibration, and how the model is implemented. Finally, Section 6 provides conclusions about the current state of research and suggests further steps.

2 BACKGROUND

System Dynamics (SD) modeling was originally developed at MIT to solve socio-economic and socio-technical problems [4]. In its essence are the ideas of *systems thinking* [5]. In systems thinking, socio-economic or socio-technical systems are represented as feedback structures whose complex behaviors are a result of interactions of many (possibly non-linear) feedback loops over time [6]. 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 [7], [8], [9], [10], [11]. Examples of its application in software engineering are evaluating different process variations, 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 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 artifacts, 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 artifact development, defect generation and allocation of personnel.

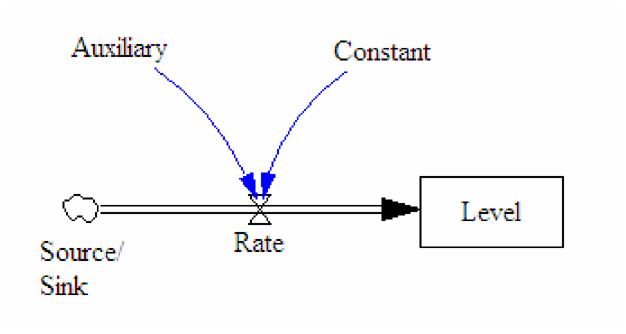

Figure 1: Schematic notation of a Rate and Level System

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.

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

Equation 1: Mathematical representation of a Rate and Level system

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

Auxiliaries are variables that are used for intermediate calculations, e.g., the portion of a software artifact 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 artifact 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*.

3 Related Work

The idea of using software process simulators for predicting project performance or evaluating processes is not new. Beginning with pioneers like Abdel-Hamid [7], Bandinelli [12], Gruhn [13], Kellner [14], Scacchi [15], and many others[1], dozens of process simulation models have been developed for various purposes. However, all known models have at least one of the following shortcomings:

[1] For an overview of software process simulation works done in the past 15 to 20 years refer to [16]. Currently, a systematic review is being conducted that will offer a more comprehensive overview of work done in the field of software process simulation [17].

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 [3], 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 which makes it unsuitable for more comprehensive analysis 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 models with the second shortcoming mentioned above is reported in [18]. The goal of building the model described in [18] 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 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 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 affects 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 public and therefore it cannot be reused by others. 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 [19] 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® [20] and Vensim® [21], the fourth issue has not yet been satisfactory resolved, neither by researchers proposing proprietary process simulation modeling environments (e.g., Little-Jil [22]) nor by researchers using commercial process simulation environments.

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 [5]. He identified ten "Systems Archetypes", i.e., generic process structures which embody typical recurring behavioral patterns of individuals and organizations. "Limits to growth" is an example 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". For example growth in sales demand in a certain production organization leads to a growth in its production while 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 and more material. In this situation, availability of new workforce and excessive material time leads are among the factors that "slow down the success". Although these archetypes are certainly a good tool for understanding individual and organizational behavior modes, they are too generic and qualitative as to be directly applicable for the modeling of software development processes.

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 behavior 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. At the root of the hierarchy are the *Elements* which are the smallest individual pieces in a System Dynamics model. Below *Elements* are the *Generic flow processes* which are small microstructures consisting of only a few *Elements*. Underneath *Generic flow processes* are the *infrastructures* that are comprised of several microstructures producing more complex behaviors and finally are the *flow chains* that are *infrastructures* that include a series of *Elements* that usually form a basic "backbone" of a model portion. These 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 modelers 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 paper 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 behavioral aspects of software development processes, GENSIM 2.0 provides a blueprint on how to integrate detailed product and resource models. In GENSIM 2.0, each instance of a process artifact type and resource type, i.e., roles involved in software development, is modeled individually. GENSIM 2.0 is the core element of a long-term research program supporting the integration of results from empirical software engineering research conducted worldwide.

4 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], [16], [26]) referred to as macro-patterns. GENSIM 2.0 is an example of a process simulation model constructed from macro-patterns. This section 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.

4.1 GENERIC PROCESS STRUCTURES (MACRO-PATTERNS)

The left-hand side of Figure 2 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 artifacts are verified (e.g., inspected) right after they are developed. However, since not in all software development projects all artifacts are verified, this activity is optional.

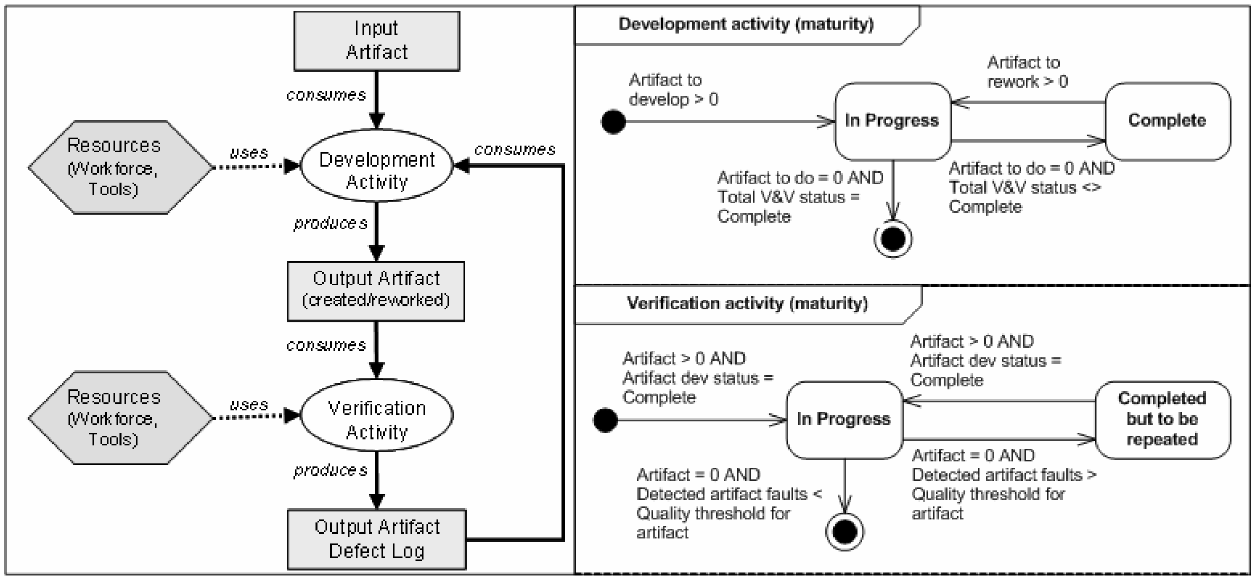
Input Artifact
consumes
Resources (Workforce, Tools)
uses
Development Activity
consumes
produces
Output Artifact (created/reworked)
consumes
Resources (Workforce, Tools)
uses
Verification Activity
produces
Output Artifact Defect Log
Development activity (maturity)
Artifact to develop > 0
In Progress
Artifact to rework > 0
Complete
Artifact to do = 0 AND Total V&V status = Complete
Artifact to do = 0 AND Total V&V status <> Complete
Verification activity (maturity)
Artifact > 0 AND Artifact dev status = Complete
In Progress
Artifact > 0 AND Artifact dev status = Complete
Completed but to be repeated
Artifact = 0 AND Detected artifact faults < Quality threshold for artifact
Artifact = 0 AND Detected artifact faults > Quality threshold for artifact

Figure 2: Macro-pattern for development/verification activity pairs (with state-transition charts)

Associated with activities are input/output products and resources. It is assumed that every development activity has some input artifacts and cannot be started if those artifacts are not ready. For example, the code development activity may not be started without the related design artifacts in place. Outputs of the development activities which are software artifacts are then 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 artifacts.

In addition, each artifact, activity, and resource is characterized by attributes representing states. *Learning* is an example attribute related to resources such as workforce. For example, 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 Fig. 2 shows 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 artifact that has to be developed becomes greater than zero, i.e., input artifacts of the development activity are ready and development can be started, the development activity transitions into the *In Progress* state. After development of an artifact is finished, if verification has to be carried out, the artifact is handed to the verification team and the development activity transitions into the *Complete* state. The same transition happens when rework of artifacts is finished and they are handed to validation teams for testing activities. Hence, in the diagram state transitions of the development activity is 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 artifact has to be reworked. This transition happens similarly in the situation where validation activities are finished and artifacts have to be reworked as a result. Whenever the development activity of an artifact is finished and no more verification and validation has to be carried out the development activity of the artifact 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 artifact 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 artifact 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 3 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 artifacts, i.e., software code or specification artifacts depending on management policies, are input to any test case development activity. The test case development activity cannot begin if these artifacts 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 artifacts, i.e., software code artifacts 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 3 shows the state transition diagrams specifying the maturity states of the test case development (top) and validation (bottom) activities. It can be seen that the test case development activity transitions into the *In Progress* state whenever software code or specification artifacts are available. Determining whether or not test cases can be derived directly from the specification artifacts before the code artifacts are ready, i.e., *Specification artifact* 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 artifacts the test case development activity is finalized.

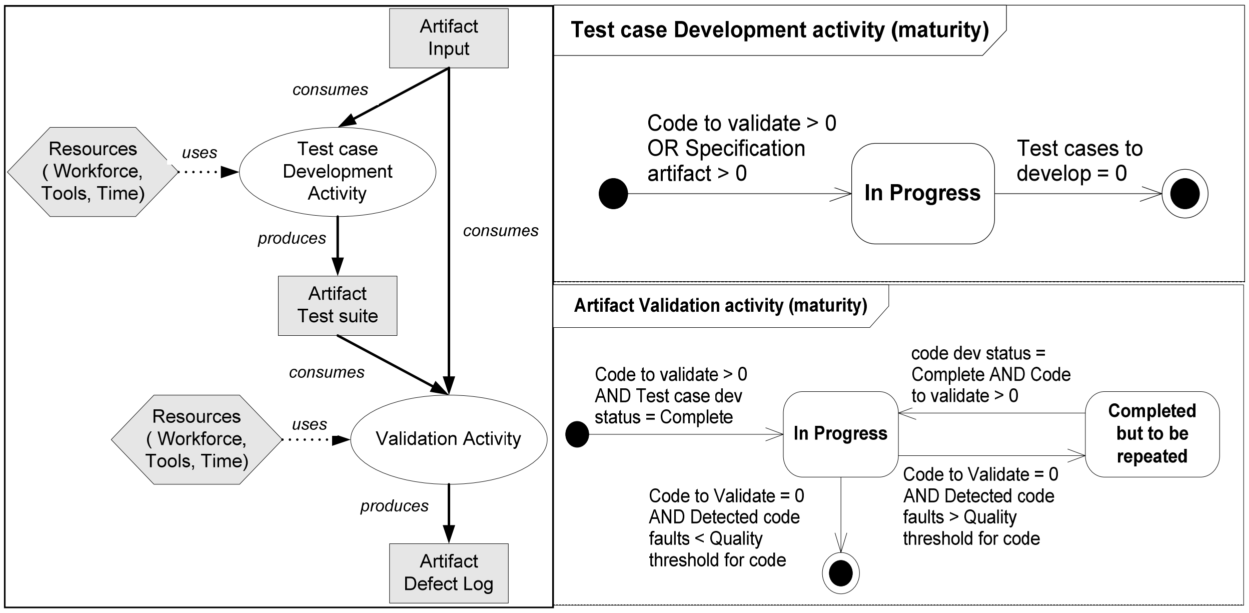
Artifact Input
consumes
Resources (Workforce, Tools, Time)
uses
Test case Development Activity
produces
consumes
Artifact Test suite
consumes
Resources (Workforce, Tools, Time)
uses
Validation Activity
produces
Artifact Defect Log
Test case Development activity (maturity)
Code to validate > 0 OR Specification artifact > 0
In Progress
Test cases to develop = 0
Artifact Validation activity (maturity)
Code to validate > 0 AND Test case dev status = Complete
In Progress
code dev status = Complete AND Code to validate > 0
Completed but to be repeated
Code to Validate = 0 AND Detected code faults > Quality threshold for code
Code to Validate = 0 AND Detected code faults < Quality threshold for code

Figure 3: Test case development/Validation macro-pattern

The code artifact validation activity transitions into the *In Progress* state whenever code artifacts that have to be validated are available and the required test cases are developed. Whenever all of the code artifacts 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 artifacts 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 artifacts is finished and they become available for validation.

4.2 THE V-MODEL DEVELOPMENT PROCESS

For the current implementation of GENSIM 2.0, the macro-patterns shown in Figure 2 and 3 are employed to represent an instance of the well-known V-Model software development process shown in Figure 4. As described in the development/verification macro-pattern discussed above 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 artifacts are captured as instances of different software artifacts types. For example, a specific design artifact of a subsystem is an instance of the artifact type *design artifact*. Software artifacts types are associated with one of three granularity or refinement levels of software development as follows:

- **System Level** includes activities carried out on artifacts 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 artifacts 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 artifacts 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 artifacts 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 views 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 (V&V) 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 no matter by which activity defects are found, while rework of design and requirements artifacts 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 Fig. 4 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:

1. **Product Flow View** models the specifics of how software artifacts are processed (developed, reworked, verified and validated) and sent back and forth during and between different activities of the development project.
2. **Defect Flow View** models the specifics of how defects are moved around (generated, propagated, detected and corrected) as different software artifact 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 artifacts as they are processed in different activities.
3. **Resource Flow View** models the specifics of how different resources (developers, techniques/tools) are allocated to different activities of the development project.
4. **State Flow View** models the specifics of how the states of different entities as explained in Section 4.1 change during the development project.

As mentioned earlier, different software artifacts 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 artifacts. Therefore if one system consists of several sub-systems, and each sub-system of several modules, then each of the individual software artifacts belonging to any of the subsystems or modules is identifiable using the subsystem's or module's subscript value.

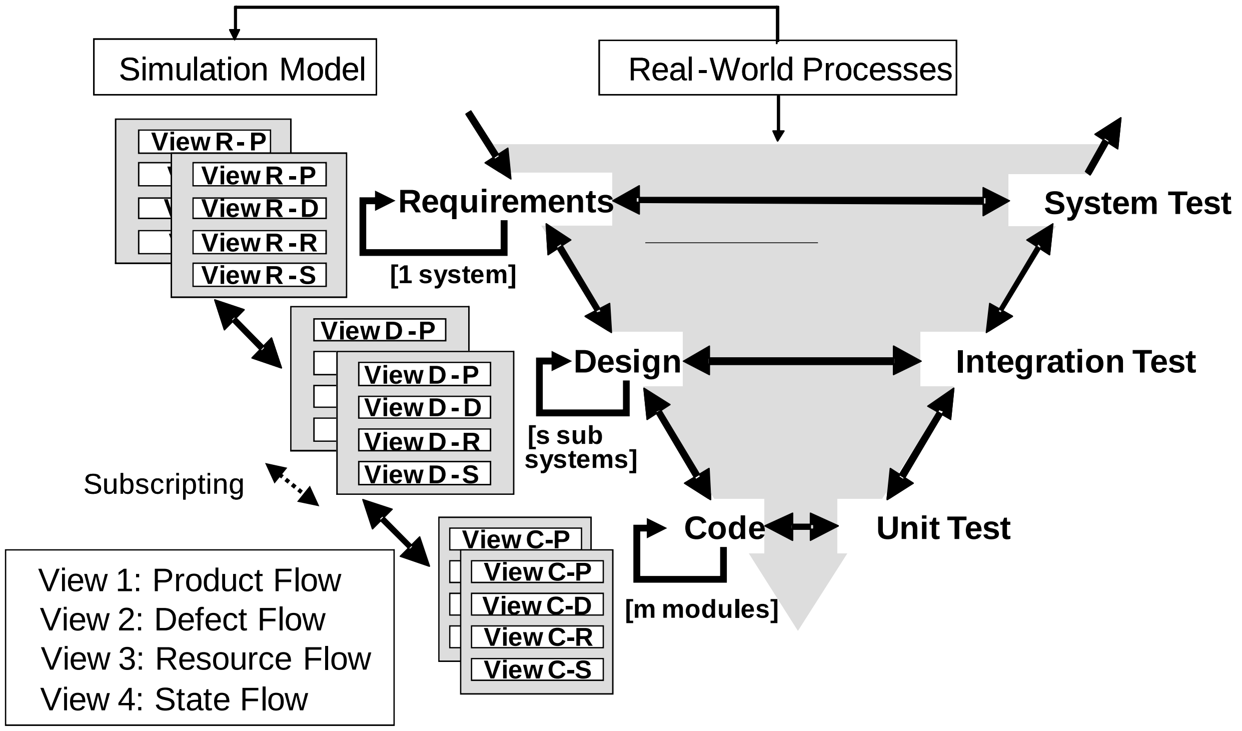

Figure 4: Application of macro-patterns to simulate the V-Model in GENSIM 2.0

5 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 Fig. 4. Having multiple views adds to the understandability and hence reusability of the model, while enabling the modeler to focus on one specific aspect at a time. Views are discussed in more detail in Section 5.2.

The subscripting mechanism provided by Vensim® is used to model individual software artifacts. 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 5.3.

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 5.4.

5.1 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 4.1 is employed in modeling all the three development phases, a similar set of parameters has been defined in all of them. As an example, in the code phase model, the level variable *Code to do size* is used to represent the amount of code artifact that is waiting to be developed. Meanwhile, a level variable called *Design to do size* is defined in the design phase model to specify the amount of a design artifact that is waiting to be developed. The same rule applies for the group 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 activities code development and verification). 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 5.2. For a complete description and all details of all the parameters and equations please refer to Appendix A.

Input parameters represent project specific information such as estimated product sizes and developer skills, as well as project specific policies that define which verification and validation activities should be performed and whether requirements and design artifacts should be reworked if defects are found in code by CI (code inspection), UT (unit test), IT (Integration test), or ST (system test) that actually originate from design or requirements defects.

Calibration parameters represent organization specific information that typically is retrieved from measurement programs and empirical studies. For a detailed description of how GENSIM 2.0 calibration is done, refer to [27].

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), end effort (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 which 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 artifacts. 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 artifacts.

The last group of parameters is the one that relates to 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.

The 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 Parameters used in modeling the code phase

	Parameter Name	**Type**	**Attribute**	**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 5.2. For a complete description and all details of all the parameters and equations refer to Appendix A.

Table 2: Important parameters used in modeling the system test phase

	Parameter Name	Type	Attribute	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

5.2 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.

5.2.1 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 4 are discussed in more detail. Since the macro-pattern discussed in Section 4.1 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 artifacts 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 artifacts to other phases, i.e., the code phase. Therefore, in the following only the code phase is explained in full detail.

5.2.1.1 Code Phase Product Flow View

The code phase product flow view captures the specifics of how code artifacts are developed, reworked and verified and sent back and forth during and between the code phase and validation phases of the development project. Figure 5 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.

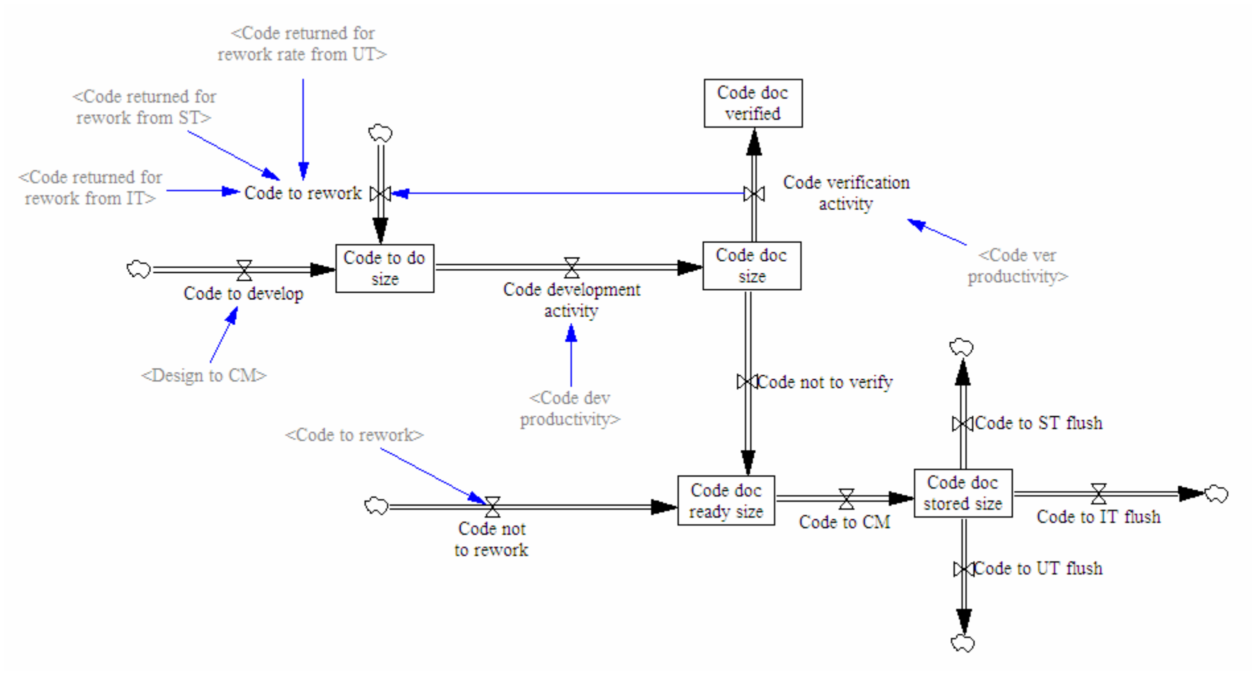

Figure 5: Code Phase Product Flow View

Software artifacts that flow through this part of the model are code artifacts of different modules of the system. It is assumed that code development for a module can only begin when the design artifacts 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 artifacts that has to be developed. These artifacts 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 artifacts are developed and then stored in the *Code doc size* level variable.

Whenever the development activity for a module's code artifact is finished, it is either verified or not according to state variables discussed in Section 5.2.1.4. If the code artifacts 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 artifact, the code artifact is sent back for rework using the *Code to rework* rate variable. This rate is also used to specify the amount of code artifact returned for rework from the validation phases (i.e. unit, integration and system test).

If at the end of the development activity the code artifact 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 artifact 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 artifact 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 artifact 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 artifact to different validation phases.

5.2.1.2 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 artifacts 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 artifacts as they are processed in the code phase. Figure 6 shows a simplified snapshot of this view with many of its auxiliary variables hidden to improve readability and understandability of the graph.

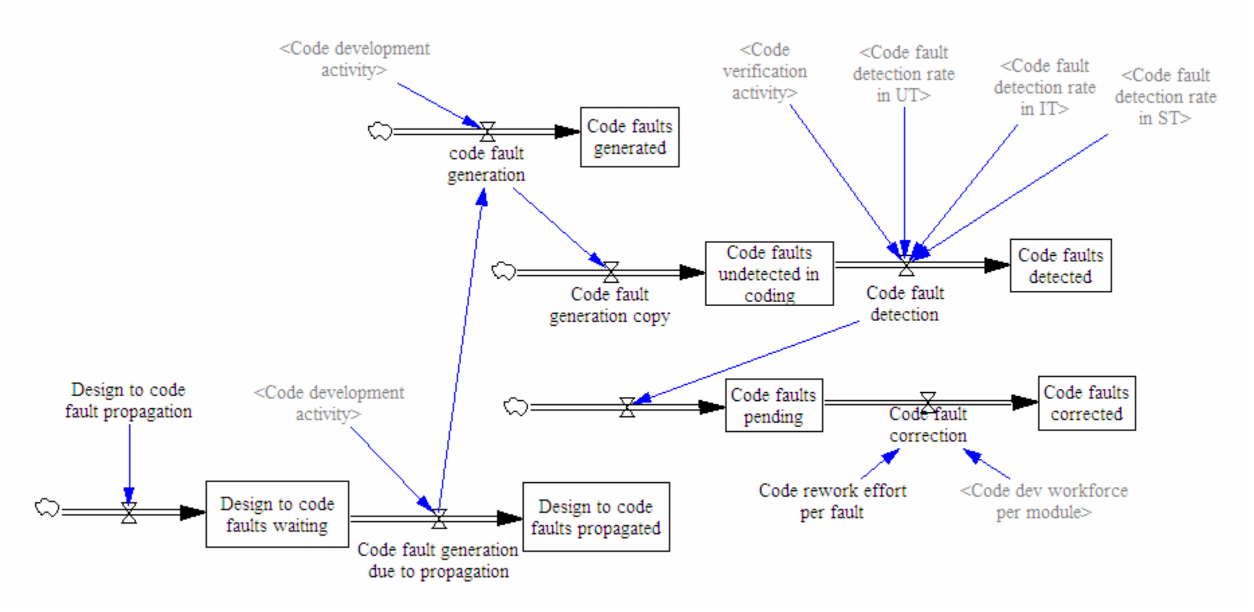
<Code development activity>
code fault generation
Code faults generated
<Code verification activity>
<Code fault detection rate in UT>
<Code fault detection rate in IT>
<Code fault detection rate in ST>
Code fault generation copy
Code faults undetected in coding
Code fault detection
Code faults detected
Design to code fault propagation
<Code development activity>
Code faults pending
Code fault correction
Code faults corrected
Design to code faults waiting
Code fault generation due to propagation
Design to code faults propagated
Code rework effort per fault
<Code dev workforce per module>

Figure 6: Code Phase Defect Flow View

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 artifact for two reasons. Firstly, there are the faults that are injected in the code artifact due to design faults in the design artifact 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 artifact 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 artifact because of the propagation.

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 artifact 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 artifacts.

5.2.1.3 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 7 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 6, it is used to determine the headcount of developers assigned to code development and verification and their skill level average. Details on exactly how *Actual Allocation* works is discussed in Section 5.4.

It is assumed that the skill level of a developer 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 [27]).

Code ver effectiveness is a constant used to represent the effectiveness of the code verification technique in detecting the code faults in the code artifact, if used by ”optimally skilled” personnel. 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 developers 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 developers.

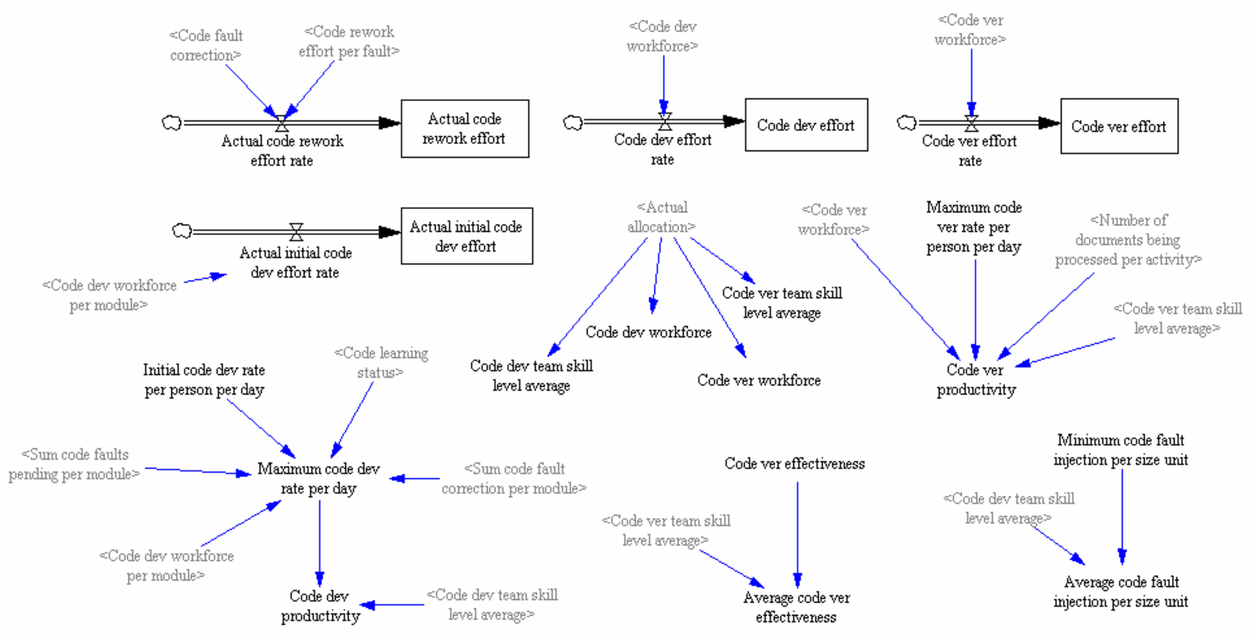
<Code fault correction>
<Code rework effort per fault>
Actual code rework effort rate
Actual code rework effort
<Code dev workforce>
Code dev effort rate
Code dev effort
<Code ver workforce>
Code ver effort rate
Code ver effort
Actual initial code dev effort rate
Actual initial code dev effort
<Code dev workforce per module>
<Actual allocation>
<Code ver workforce>
Maximum code ver rate per person per day
<Number of documents being processed per activity>
Code ver team skill level average
Code dev workforce
<Code ver team skill level average>
Code dev team skill level average
Code ver workforce
Code ver productivity
Initial code dev rate per person per day
<Code learning status>
<Sum code faults pending per module>
Maximum code dev rate per day
<Sum code fault correction per module>
Code ver effectiveness
Minimum code fault injection per size unit
<Code dev team skill level average>
<Code ver team skill level average>
<Code dev workforce per module>
Code dev productivity
<Code dev team skill level average>
Average code ver effectiveness
Average code fault injection per size unit

Figure 7: Code Phase Resource Flow View

Minimum code fault injection per size unit is a constant used to represent the number of faults that ″optimally skilled″ developers commit in the code artifact. If the skill level average of developers 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 6 it depends on the headcount of the workforce allocated to carry out the verification activity, the number of code artifacts that have to be verified i.e. the number of modules that their code artifact 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" developers can verify everyday.

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.

5.2.1.4 Code Phase State Flow View

This view captures the specifics of how the states of different entities as explained in Section 4.1 change during the code phase of the development project. Figure 8 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 artifact 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.

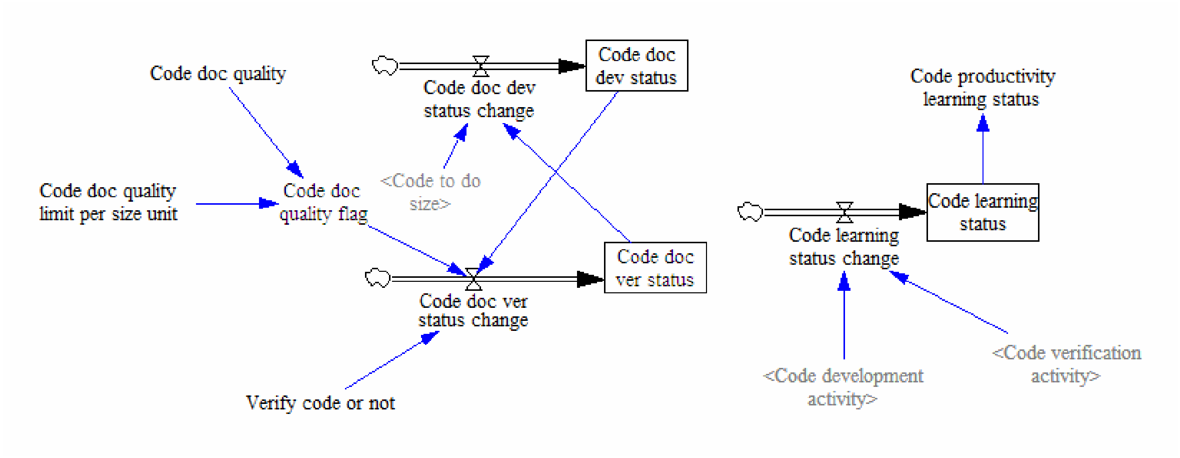
Code doc quality
Code doc dev status change
Code doc dev status
Code productivity learning status
Code doc quality limit per size unit
Code doc quality flag
<Code to do size>
Code learning status
Code learning status change
Code doc ver status
Code doc ver status change
Verify code or not
<Code development activity>
<Code verification activity>

Figure 8: Code Phase State Flow View

If *Code doc quality* (the number of code faults detected during the verification activity) is greater than 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 artifact 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 and verified. Every time the code artifact is processed, 1 is added to the value of this level. *Code productivity learning status* specifies the effect *Code learning status* on the productivity of developers.

5.2.2 Validation Views (Key Assumption/levels/rates/variables)

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 4 are discussed in more detail. Since the second macro-pattern explained in Section 4.1 is applied to 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.

5.2.2.1 System Testing Product Flow View

This view captures the specifics of how code artifacts are validated and sent back and forth between system testing validation and code phases of the development project. Figure 9 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.

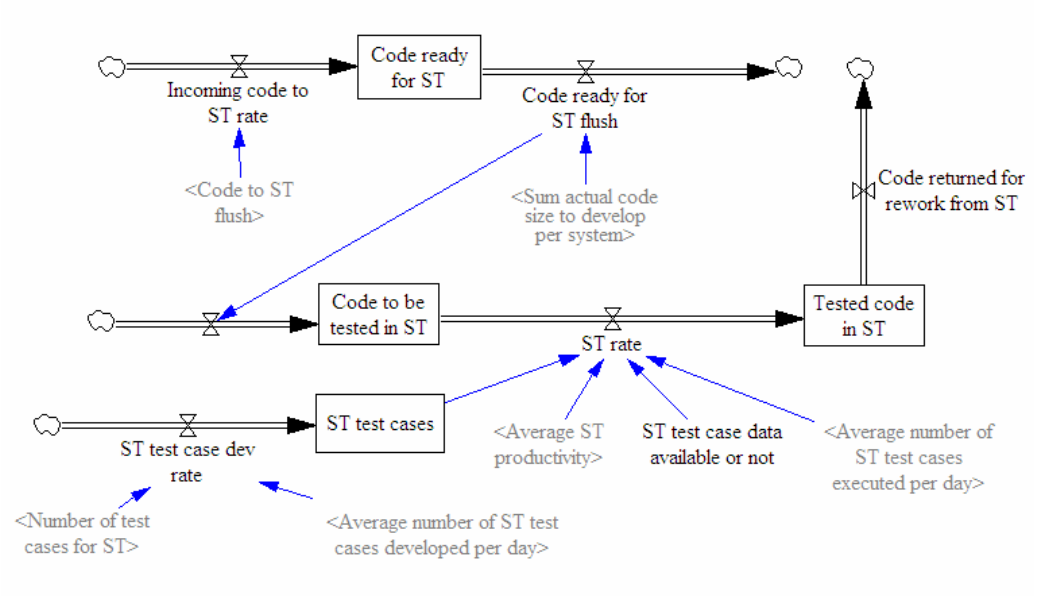
Code ready for ST
Incoming code to ST rate
Code ready for ST flush
<Code to ST flush>
<Sum actual code size to develop per system>
Code returned for rework from ST
Code to be tested in ST
Tested code in ST
ST rate
ST test cases
ST test case dev rate
<Average ST productivity>
ST test case data available or not
<Average number of ST test cases executed per day>
<Number of test cases for ST>
<Average number of ST test cases developed per day>

Figure 9: System test Product Flow View

It is assumed that code artifacts 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 artifacts 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 artifacts that are ready for system testing. The *Code ready for ST* is emptied and all code artifacts are moved to *Code to be tested in ST* (Using the *Code ready for ST flush* rate variable) whenever code artifacts 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 artifacts of the system) are waiting to be system tested.

ST test case data available or not is a constant 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 for it and the number of developed test cases (represented using the *ST test cases* level variable) reaches the number of required system test cases for the system (represented using the *Number of test cases for ST* variable). It is assumed 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 artifacts 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.

5.2.2.2 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 10 shows a simplified snapshot of this view with many of its auxiliary variables hidden to improve readability and understandability of the graph.

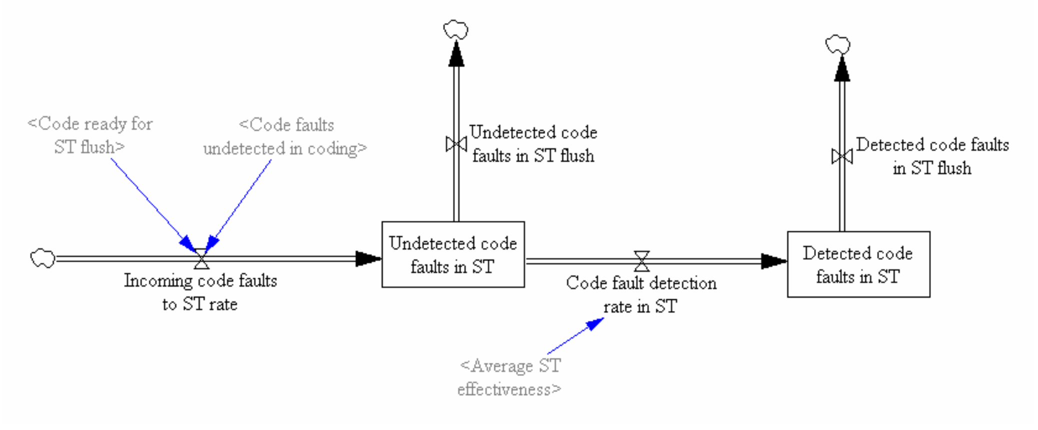
<Code ready for ST flush>
<Code faults undetected in coding>
Undetected code faults in ST flush
Detected code faults in ST flush
Incoming code faults to ST rate
Undetected code faults in ST
Code fault detection rate in ST
Detected code faults in ST
<Average ST effectiveness>

Figure 10: System testing Defect Flow View

Incoming code faults to ST rate is used to propagate all the code faults existing in the system code artifacts (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 code faults in the system 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 reset as well 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.

5.2.2.3 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 11 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.

If system test case calibration data is available, it is assumed that the average skill level of the workforce who 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 test case development for system testing. *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 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.

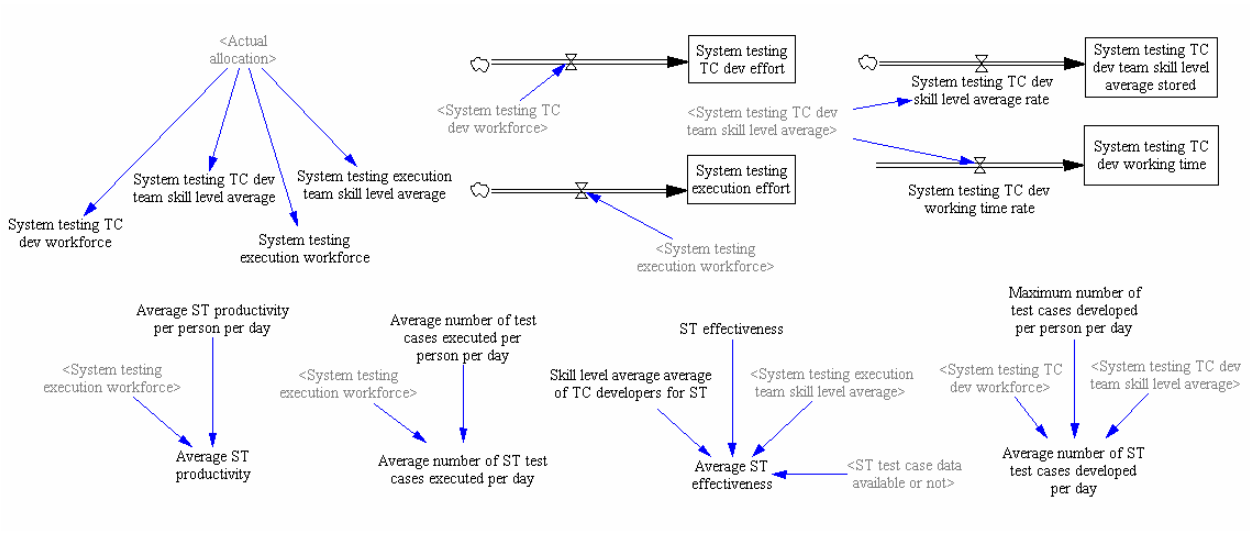
<Actual allocation>
System testing TC dev effort
System testing TC dev team skill level average stored
<System testing TC dev workforce>
<System testing TC dev team skill level average>
System testing TC dev skill level average rate
System testing TC dev working time
System testing TC dev team skill level average
System testing execution team skill level average
System testing execution effort
System testing TC dev working time rate
System testing TC dev workforce
System testing execution workforce
<System testing execution workforce>
Maximum number of test cases developed per person per day
Average ST productivity per person per day
Average number of test cases executed per person per day
ST effectiveness
<System testing execution workforce>
<System testing execution workforce>
Skill level average average of TC developers for ST
<System testing execution team skill level average>
<System testing TC dev workforce>
<System testing TC dev team skill level average>
Average ST productivity
Average number of ST test cases executed per day
Average ST effectiveness
<ST test case data available or not>
Average number of ST test cases developed per day

Figure 11: System Test Resource Flow View

If system test case calibration data is available, the productivity of system testing, i.e., the amount of code artifact 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 rather an automated procedure of running the test cases and reporting the results.

5.2.2.4 System testing State Flow View

This view captures the specifics of how the states of different entities as explained in Section 4.1 change during the system test phase of the development project. Figure 12 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 test cases. If value of this constant is set to 1, system test case development begins only when all the code artifacts of the system are ready and if set to 0, system test case development can begin as soon as the requirements specification artifacts of the system are completed and verified.

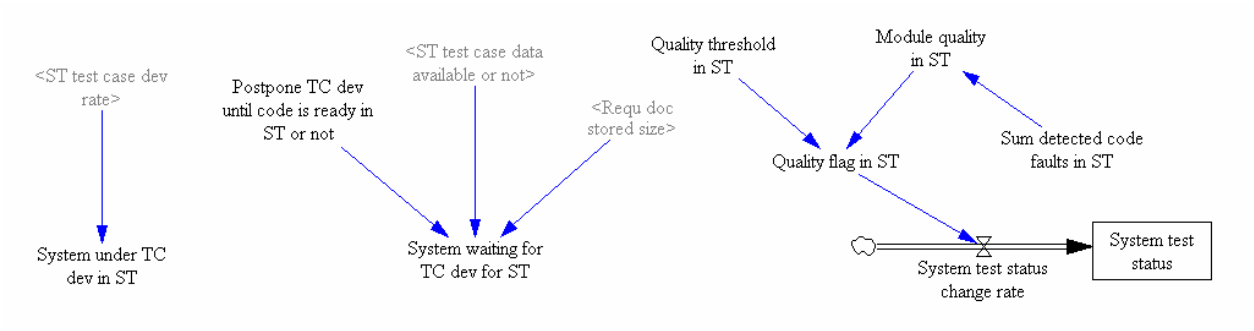
<ST test case dev rate>
System under TC dev in ST
Postpone TC dev until code is ready in ST or not
<ST test case data available or not>
<Requ doc stored size>
System waiting for TC dev for ST
Quality threshold in ST
Module quality in ST
Quality flag in ST
Sum detected code faults in ST
System test status change rate
System test status

Figure 12: 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.

5.3 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 it 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 $MOD1, MOD2, \dots, MOD100$ to represent 100 modules within the system. 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 $SUB1, SUB2, \dots, SUB5$ to model 5 different subsystems in the system. 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. By a defect origin it is meant the phase in which the fault was actually committed. Its subscript values are currently specified as requ, design and code representing requirements specification, design and code phases respectively. It is generally associated with variables used to model faults of software artifacts.

Factor is a subscript used to identify different aspects of software quality that faults have an effect on. By aspects of software quality it is meant software quality characteristics as identified in the ISO 9126 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 artifacts could be characterized and differentiated by the quality aspect that they have the most effect on. Like the *Origin* subscript, it is generally associated with variables that model faults of software artifacts and can be modified easily to enable evaluation of even more different aspects of quality.

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

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.

5.4 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 main allocation function takes as input headcount and skill levels of the available workforce, workload of different activities and the minimum skill level required for developers 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 1, 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 ith developer in carrying out the jth 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.

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

Equation 2: Skill level Matrix S

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 jth activity. The value for w_j is determined by the number of artifacts, 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 jth activity.

To prepare the allocation of developers 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 ith developer can be assigned to carry out the jth activity having at least the activity's required skill level.

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

- Step 1: assign to activity j all developers that can only carry out the jth activity
- Step 2: assign to activity j a portion of the developers which can carry out the jth activity and exactly one of the other activities for which $w_j > 0$
- Step 3: assign to activity j a portion of the developers which can carry out the jth activity and exactly 2 of the other activities for which $w_j > 0$
- $\vdots$
- Step k: assign to activity j a portion of the developers which can carry out the jth activity and exactly 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 developers which will be assigned to the jth activity in the tth step for any of the possible permutations is determined using the formula shown in Equation 2. As the following calculations may result in floating point numbers, results are rounded if necessary.

$$P_j = \frac{w_j}{w_j + \sum_i w_i} \quad and \quad i \in \{t-1 \ activities other than \ j\}$$

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

6 CONCLUSION AND FUTURE WORK

GENSIM 2.0 is a complex publicly available customizable software process simulation model. 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 by exploiting the subscription mechanisms of Vensim. Moreover, the possibility to use external DLL libraries gives the opportunity to extract potentially time-consuming algorithms from the SD model and thus speed up model execution.

Future work on GENSIM 2.0 will address some of its current limitations. Currently it is not possible to represent incremental software development processes easily. Mechanisms will be added to the model that allow for concurrent execution of development cycles following the generic process shown in Fig. 4. 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 works on GENSIM 2.0 will address this issue and implement mechanisms for dealing with changeable workforce profiles. 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 is part of a long-term research program that aims at combining results from empirical studies and company-specific measurement programs with process simulation. Currently, GENSIM 2.0 is calibrated to data received from a German research institute and its industrial partners. The calibrated model will be used to explore which combination of V&V activities and techniques is most suitable to achieve certain product quality goals defined according to standard ISO 9126, under given resource and time constraints.

REFERENCES

[1] M. I. Kellner, R. J. Madachy, and D. M. Raffo, " Software process simulation modeling: Why? What? How?," *Journal of Systems and Software,* vol. 46, pp. 91-105, 1999.

[2] H. Waeselynck and D. Pfahl, "System Dynamics Applied To The Modeling Of Software Projects," *Software Concepts and Tools,* vol. 15, pp. 162-176, 1994.

[3] 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, pp. 283-298, 2001.

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

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

[6] G. P. Richardson, *Feedback Thought in Social Science and Systems Theory*: University of Pennsylvania Press, 1990.

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

[8] 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.

[9] R. J. Madachy, "A software project dynamics model for process cost, schedule and risk assessment," University of Southern California, 1994, p. 127.

[10] 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.

[11] T. John Douglas, "An extensible model for evaluating the impact of process improvements on software development cycle time," Arizona State University, 1996, p. 386.

[12] 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, pp. 440-454, 1995.

[13] G. Volker and S. Armin, "Software Process Validation Based on FUNSOFT Nets," in *Proceedings of the Second European Workshop on Software Process Technology*: Springer-Verlag, 1992.

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

[15] 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, pp. 283-289, 1990.

[16] M. Muller and D. Pfahl, "Simulation Methods," in *Guide to Advanced Empirical Software Engineering*: Springer London, 2008, pp. 117-152.

[17] Z. He, K. Barbara, and P. Dietmar, "Reflections on 10 Years of Software Process Simulation Modeling: A Systematic Review," in *ICSP* Leipzig, Germany, 2008.

[18] 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, pp. 197-205, 2004.

[19] 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, pp. 487-510, 2000.

[20] ExtendSim, "http://www.imaginethatinc.com/", March 2008.

[21] Vensim, "http://www.vensim.com/", Feb. 2008.

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

[23] R. Madachy, "Reusable Model Structures and Behaviors for Software Processes," in *Software Process Change*. vol. 3966/2006: Springer Berlin / Heidelberg, pp. 222-233.

[24] N. Angkasaputra and D. Pfahl, "Making Software Process Simulation Modeling Agile and Pattern-based," in *ProSim 2004*, 2004, pp. 222-227.

[25] O. Armbrust, T. Berlage, T. Hanne, P. Lang, J. Mí¼nch, 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, 2005, pp. 333-364.

[26] D. Raffo, G. Spehar, and U. Nayak, "Generalized Process Simulation: What, Why and How?," in *ProSim '03 Workshop*, 2003.

[27] K. Khosrovian, D. Pfahl, and V. Garousi, "Calibrating a Customizable System Dynamics Simulation Model of Generic Software Development Processes," Schulich School of Engineering, University of Calgary SERG-2007-08, 2008.

APPENDIX A- GENSIM 2.0 EQUATIONS

This appendix includes the complete set of all GENSIM 2.0 parameters along with their equations. Due to application of macro-patterns in the modeling of development/verification phases, analogous sets of parameters have been used for implementing each of these phases. Hence, the descriptions are provided for only one of them, i.e., requirements specification and analogous descriptions could be defined for analogous parameters in the other phases. In cases of any exceptions, i.e., parameters used in the implementation of the design or code phases do not have corresponding parameters in the requirements specification phase, descriptions are provided individually. The same rule applies for validation phases and the descriptions are given only for the unit test phase.

```
********************************************************
	.Requirement Specification Process
********************************************************

Requ doc stored flush due to code rework=
	Actual requ size to specify*SUM(Portion of code reworked to actual system code size per subsystem\
		[subsystem!])
	~	Page/Day
	~	This variable shows the number of pages of the requirements specification \
		document which has to be reworked per day because of code defects detected \
		during code verification or unit/integration/system test which originate \
		in requirement specification.
	|

Requ late rework incoming rate=
	IF THEN ELSE(Portion of requ spec to rework*TIME STEP>Requ doc stored size,Portion of requ spec to rework\
		,0)
	~	Page/Day
	~	Because sometimes a page of the requirements document has to reworked \
		because of some detected defects but it is already being reworked because \
		of some other defect detected in an upstream phase, this rate is used to \
		specify incoming rate of the amount of this kind of rework.
	|

Requ late rework outgoing rate=
	IF THEN ELSE(Portion of requ spec to rework*TIME STEP<Requ doc stored size,MIN(Requ to be reworked later\
		/TIME STEP,Requ doc stored size/TIME STEP-Portion of requ spec to rework),0)
	~	Page/Day
	~	This rate is used to send back the number of pages of the requirements \
		specification document which are waiting to be sent for rework in the \
		'Requ to be reworked later' level for rework as soon as they are stored in \
		the 'Requ doc stored size' level.
	|

Requ doc stored flush=
	IF THEN ELSE(Rework requ spec or not=1:AND:Portion of requ spec to rework*TIME STEP<\
		Requ doc stored size,Portion of requ spec to rework
	,0)
	~	Page/Day
	~	This rate specifies the number of pages of the requirements specification \
		document which is sent back for rework everyday.
	|

Amount of requ spec reworked= INTEG (
	Requ spec to rework,
		0)
	~	Page
	~	This level variable keeps track of the number of requirements \
		specification document which are reworked. Everytime a page is reworked, \
		one page is added to this variable.
	|

Portion of requ spec to rework=
	Requ doc stored flush due to code rework+Requ doc stored flush due to design verification
	~	Page/Day
	~	This variable shows the total number of pages of the requirements \
		specification document which has to be reworked everyday.
	|
Requ to be reworked later= INTEG (
        Requ late rework incoming rate-Requ late rework outgoing rate,
                0)
        ~       Page
        ~       This level variable is used to keep track of the number of pages of the \
                requirements specification that has to be reworked due to some detected \
                defect but is already being reworked.
        |

Requ doc stored flush due to design verification=
        IF THEN ELSE(SUM(Actual design size to develop[subsystem!])>0,Actual requ size to specify\
                *Design to rework due to verification/SUM(Actual design size to develop[subsystem!]\
                ),0)
        ~       Page/Day
        ~       This variable shows the number of pages of the requirements specification \
                document which has to be reworked per day because of design defects \
                detected during design verification which originate in requirement \
                specification.
        |

Requ doc stored late flush=
        Rework requ spec or not*Requ late rework outgoing rate
        ~       Page/Day
        ~       This rate is equal to the 'Requ late rework outgoing rate' if the \
                requirements specification document has to be reworked.
        |

Rework requ spec or not=
        1
        ~       Dmnl
        ~       This constant is used to switch on/off reworking the requirements \
                specification document due to design/code defects detected with \
                requirement specification origin during downstream V&V activities. while \
```

set to 1 the requirement specification document will be reworked and while \
set to 0 it will not be reworked.

|

Requ to CM=

IF THEN ELSE(Requ doc ready size>Average requ size in pages-0.001:AND:Total design doc dev status\
=0,Requ doc ready size/TIME STEP
,IF THEN ELSE(Total design doc dev status>0,Requ doc ready size/TIME STEP,0))

~ Page/Day

~ This rate specifies the number of pages of the requirements specification \
document that are ready to be stored in the configuration management \
system everyday

|

Requ spec not to verify=

IF THEN ELSE(Requ spec ver status>=3,Requ spec size/TIME STEP,IF THEN ELSE(Requ spec ver status\
=3:AND:Requ spec status=2, Requ spec size/TIME STEP
,IF THEN ELSE(Verify requ spec or not=0,Requ spec size/TIME STEP,0)))

~ Page/Day

~ This rate specifies the number of pages of the requirements specification \
document which do not need to be verified everyday.

|

Requ spec verified flush=

IF THEN ELSE(Requ spec status=1,Requ spec verified/TIME STEP,0)

~ Page/Day

~ This rate is used to reset the 'Requ spec verified' level variable at the \
end of the verification activity.

|

Requ specification activity=

IF THEN ELSE(Requ spec status=1,MIN(Requ spec to do size/TIME STEP,Average requ spec dev rate per day\

),0)

~ Page/Day

~ This rate shows the number of pages of the requirements specification \
document which are worked on every day.

|

Maximum requ spec rate per day=

IF THEN ELSE(Requ spec learning status>1:AND:Sum requ spec faults pending>0,Requ spec to do size\

*Sum requ spec fault correction rate/Sum requ spec faults pending, Initial max requ spec rate per person and day\

*Requ dev workforce)

~ Page/Day

~ This variable specifies the maximum number of pages of the requirements \
specification document that developers can develop/rework everyday. By \
maximum it is meant that developers that their average skill level is 1 \
will develop/rework this number of pages everyday.

|

Average requ spec dev rate per day=

Maximum requ spec rate per day*Requ dev team skill level average

~ Page/Day

~ This variable specifies the number of pages of the requirements \
specification document that developers can develop/rework everyday.

|

Requ spec to rework=

IF THEN ELSE(Requ spec doc quality ratio>0,Requ spec verification activity*MIN(1,Requ spec doc quality ratio\

)+Requ doc stored flush+Requ doc stored late flush

,Requ spec verification activity+Requ doc stored flush+Requ doc stored late flush)

~ Page/Day

~ This rate shows the incoming rate of the requirements specification \
document for rework.

|

```
Requu doc stored size= INTEG (
        Requ to CM-Requ doc stored flush-Requ doc stored late flush,
                0)
        ~       Page
        ~       This level variable shows the number of pages of the requirements document \
                that are stored in the configuration management system.
        |

Average requ spec ver rate=
        Requ ver workforce*Maximum requ spec ver rate per peson and day*Requ ver team skill level
average
        ~       Page/Day
        ~       This variable specifies the number of pages of the requirements \
                specification document that developers can verify everyday.
        |

Requ spec verification activity=
        IF THEN ELSE(Requ spec ver status=1:OR:(Requ spec ver status=2:AND:Requ spec status=\
                2),MIN(Requ spec size/TIME STEP, Average requ spec ver rate
        ),0)
        ~       Page/Day
        ~       This rate specifies the number of pages of the requirements specification \
                document which are verified everyday.
        |

Number of test cases for ST=
        Actual requ size to specify*Average number of test cases per requ spec size unit
        ~       Testcase
        ~       This variable is used to calculate the number of test cases which need to \
                be developed to the test the whole system against the whole requirements \
                specification document
        |
Requu to specify=
	IF THEN ELSE(Time=0,Average requ size in pages/TIME STEP,0)
	~	Page/Day
	~	This rate specifies the incoming rate of the requirements specification \
		document which has to be worked on for the first time.
	|

Actual requ size to specify= INTEG (
	Requ to specify,
		0)
	~	Page
	~	This level variable is used to keep the total number of the requirements \
		specification document pages since the 'Requ spec to do size' fills and \
		empties constantly.
	|

Average number of test cases per requ spec size unit=
	5
	~	Testcase/Page
	~	This constant specifies the average number of system test cases that need \
		to be developed for testing the whole system against every page of the \
		requirements specification document.
	|

Requ spec productivity learning amplifier=
	1
	~	Dmnl
	~	This constant is used to adjust the effect of 'Requ spec productivity \
		learning status' on per defect fixing effort for requirements \
		specification defects.
	|

Requ spec verified= INTEG (
	Requ spec verification activity-Requ spec verified flush,
		0)
	~	Page
	~	This level shows the number of pages of the requirements specification \
		document which has been verified during the verification activity.
	|

Requu spec doc quality ratio=
	IF THEN ELSE(Requ spec verification activity>0:AND:Requ spec quality limit>0,Sum requ spec fault detection\
		/(Requ spec quality limit*Requ spec verification activity),0)
	~	Dmnl
	~	This ratio specifies the portion of the requirements specification \
		document that has to be reworked considering the quality threshold set for \
		this document.
	|

Requ spec not to rework=
	MAX(Requ spec verification activity-Requ spec to rework,0)
	~	Page/Day
	~	This rate specifies the number of pages of the requirements specification \
		document that does not need to be reworked considering the quality \
		threshold set for this document.
	|

Maximum requ spec ver rate per peson and day=
	8
	~	Page/(Person*Day)
	~	This constant specifies the maximum number of pages of the requirements \
		specification document that developers can verify everyday. By maximum it \
		is meant that developers that their average skill level is 1 will verify \
		this number of pages everyday.
	|

Requ spec to do size= INTEG (
	Requ spec to rework+Requ to specify-Requ specification activity,
		0)
	~	Page
	~	This level variable keeps track of the number of requirements \
		specification document which are waiting to be worked on.
	|

Requ doc ready size= INTEG (
	Requ spec not to rework+Requ spec not to verify-Requ to CM,
		0)
	~	Page
	~	This level variable keeps track of the number of pages of the requirements \
		specification document that are ready to be stored in the configuration \
		managament system.
	|

Requ spec size= INTEG (
	Requ specification activity-Requ spec not to verify-Requ spec verification activity,
		0)
	~	Page
	~	This level variable shows the number of requirements specification \
		document which has been worked on.
	|

Average requ size in pages=
	50
	~	Page
	~	This constant specifies the size of the requirements specification document
	|

********************************************************
	.Requirement Specification Quality
********************************************************~
	|
Requu spec faults detected flush[origin,factor]=
	IF THEN ELSE(Requ spec ver status>1, Requ spec faults detected[origin,factor]/TIME STEP\
		, 0)
	~	Defect/Day
	~	This rate is used to reset the 'Requ spec faults detected' level variable \
		at the end of the verification acitvity.
	|

Requ spec faults corrected flush[origin,factor]=
	IF THEN ELSE(Requ spec status=2 ,Requ spec faults corrected[origin,factor]/TIME STEP\
		, 0)
	~	Defect/Day
	~	This rate is used to reset the 'Requ spec faults corrected' level variable \
		at the end of rework activity.
	|

Maximum requ spec ver effectiveness[requ,factor]=
	Requ spec ver effectiveness constant ~~|
Maximum requ spec ver effectiveness[design,factor]=
	0,0,0 ~~|
Maximum requ spec ver effectiveness[code,factor]=
	0,0,0
	~	Dmnl
	~	This constant specifies the maximum effectiveness of the requirements \
		specification verification technique in finding undetected requirements \
		specification defects. By maximum it is meant that developers that their \
		average skill level is 1 will be able to detect this portion of the \
		undetected requirements specification defects.
	|

Requ spec fault correction rate[origin,factor]=
	IF THEN ELSE(Requ dev workforce>0,Rework requ spec or not*MIN(Requ dev
workforce/(Requ spec rework effort per fault\
		*3),Requ spec faults pending[origin,factor]/TIME STEP),0)
	~	Defect/Day
	~	This rate specifies the number of requirements specification defects which \
		are fixed everyday.
	|

requ spec fault detection rate[requ,factor]=
	IF THEN ELSE(Requ spec verification activity>0,MIN(Requ spec faults undetected[requ,\
		factor]/TIME STEP,Average requ spec ver effectiveness
	[requ,factor]*
	Requ spec verification activity*(Requ spec faults undetected[requ,factor]+Requ spec faults
detected\
		[requ,factor])/(Requ spec size
	+Requ spec verified)),SUM(Code fault detection[requ,factor,module!])/9+Sum design fault
detection due to verification\
		[requ
	,factor]/3) ~~|
requ spec fault detection rate[design,factor]=
	0,0,0 ~~|
requ spec fault detection rate[code,factor]=
	0,0,0
	~	Defect/Day
	~	This rate specifies the number of defects which are detected everyday \
		during verification of the requirements specification document.
	|

Requ spec fault generation[origin,factor]=
	Requ specification activity*Average requ spec fault injection rate per size unit[origin\
		,factor]/MAX(1,Requ spec learning status
	^Learning amplifier for requ fault injection)
	~	Defect/Day
	~	This rate specifies the number of defects that developers inject in the \
		requirements specification document during development and rework per day.
	|
Average requ spec fault injection rate per size unit[origin,factor]=
	Minimum requ spec fault injection rate per size unit[origin,factor]+Minimum requ spec fault injection rate per size unit\
		[origin,factor]*(1-Requ dev team skill level average)
	~	Defect/Page
	~	This variable specifies the number of defects that developrs inject in \
		every page of the requirements specification document.
	|

Actual requ spec faults corrected[origin,factor]= INTEG (
	Requ spec fault correction rate[origin,factor],
		0)
	~	Defect
	~	This level is used to store the overall number of defects which are \
		corrected because the 'Requ spec faults corrected' level is reset at the \
		end of rework activities.
	|

Actual requ spec faults detected[origin,factor]= INTEG (
	requ spec fault detection rate[origin,factor],
		0)
	~	Defect
	~	This level variable is used to store the number of requirements \
		specification defects detected because the 'Requ spec faults detected' \
		level variable is reset at the end of the verification.
	|

Learning amplifier for requ fault detection=
	3
	~	Dmnl
	~	This constant is used to adjust the effect of learning on detection of \
		defects during requirements specification re-verification.
	|
Requu spec fault generation copy[origin,factor]=
        Requ spec fault generation[origin,factor]
        ~       Defect/Day
        ~               |

Requ spec faults corrected[origin,factor]= INTEG (
        Requ spec fault correction rate[origin,factor]-Requ spec faults corrected flush[origin\
                ,factor],
                0)
        ~       Defect
        ~       This level variable is used to keep track of the number of requirements \
                specification defects that have been fixed during rework.
        |

Requ spec faults detected[origin,factor]= INTEG (
        requ spec fault detection rate[origin,factor]-Requ spec faults detected flush[origin\
                ,factor],
                0)
        ~       Defect
        ~       This level variable is used to keep track of the number of defects which \
                are detected during requirements specification verification.
        |

Requ spec faults pending[origin,factor]= INTEG (
        requ spec fault detection rate[origin,factor]-Requ spec fault correction rate[origin\
                ,factor],
                0)
        ~       Defect
        ~       This level variable is used to keep track of the number of requirement \
                specification defects which are pending to be fixed.
        |

Requ spec faults undetected[origin,factor]= INTEG (
        Requ spec fault generation copy[origin,factor]-requ spec fault detection rate[origin\
		,factor],
		0)
	~	Defect
	~	This level is used to keep track of the number of defects which have been \
		left undetected in the requirements specification document.
	|

Minimum requ spec fault injection rate per size unit[requ,factor]=
	13.38,13.38,13.38 ~~|
Minimum requ spec fault injection rate per size unit[design,factor]=
	0,0,0 ~~|
Minimum requ spec fault injection rate per size unit[code,factor]=
	0,0,0
	~	Defect/Page
	~	This constant specifies the minimum number of defects that developrs \
		inject in every page of the requirements specification document. By \
		minimum it is meant that developers that their average skill level is 1 \
		will inject this number of defects in every page of the requiements \
		specification document.
	|

Requ spec faults generated[origin,factor]= INTEG (
	Requ spec fault generation[origin,factor],
		0)
	~	Defect
	~	This level variable is used to store the total number of defects that are \
		injected in the requirements specification document by the developers \
		during development and rework.
	|

Sum requ spec fault detection=
	SUM(requ spec fault detection rate[origin!,factor!])
	~	Defect/Day
	~	This variable is used to show the sum of requirements specification \
		defects detected over all origins and quality factors.
	|

********************************************************
	.Requirement Specification Status
********************************************************~
	|

Requ spec ver change=
	IF THEN ELSE((Requ spec ver status=0):AND:(Requ spec status>1):AND:Verify requ spec or not\
		=1,1,IF THEN ELSE
	((Requ spec ver status=1):AND:(Requ spec size<=0):AND:(Requ spec quality flag>0):AND:\
		Verify requ spec or not=1,1,IF THEN ELSE
	((Requ spec ver status=2
	):AND:(Requ spec size>0):AND:(Requ spec status<>1):AND:Verify requ spec or not=1,-1,\
		IF THEN ELSE((Requ spec ver status=
	1):AND:(Requ spec size<=0):AND:
	(Requ spec quality flag<1):AND:Verify requ spec or not=1,2,IF THEN ELSE(Verify requ spec or not\
		=0:AND:Requ spec ver status=0,0,0)))))
	~	Dmnl/Day
	~	This rate is used to specify the rate of change in the 'Requ spec ver \
		status' level variable.
	|

Requ spec learning change=
	IF THEN ELSE(Actual requ size to specify>0,(Requ specification activity+Requ spec verification activity\
		)/Actual requ size to specify,0)
	~	Dmnl/Day
	~	This rate is used to specify the changes in the learning status of the \
		requirements specification document.
	|
Requ spec quality=
	IF THEN ELSE(Requ spec verified>0,Sum requ spec faults pending/Requ spec verified, 0\
		)
	~	Defect/Page
	~	This variable is used to specify the number of requirements specification \
		defects detected and not yet fixed in every page of the requirements \
		specification document.
	|

Requ spec quality flag=
	IF THEN ELSE(Requ spec quality limit>0:AND:Requ spec quality>Requ spec quality limit\
		,1,IF THEN ELSE(Requ spec quality limit =0,0,0))
	~	Dmnl
	~	This flag is used to show the current quality of the requirements \
		specification document is acceptable according to the 'Requ spec quality \
		limit' and 'Requ spec quality'.
	|

Requ spec learning status= INTEG (
	Requ spec learning change,
		0)
	~	Dmnl
	~	This level is used to specify the learning status of the requirements \
		specification document. Everytime this document is \
		developed/verified/reworked one is added to this value.
	|

Requ spec status change=
	IF THEN ELSE((Requ spec status=0):AND:(Requ spec to do size>0),1,IF THEN ELSE
	((Requ spec status=1):AND:
	(Requ spec to do size<=0),1,IF THEN ELSE((Requ spec status=2):AND:(Requ spec to do size\
		>0):AND:(Requ spec ver status<>1),-1,0)))
	~	Dmnl/Day
	~	This rate is used to spefiy the changes in the 'Requ spec status' level \
		variable.
	|

Requ spec quality limit=
	0
	~	Defect/Page
	~	This constant is used to specify the acceptable number of defects that can \
		be detected and fixed in every page of the requirements specification \
		document.
	|

Requ spec status= INTEG (
	Requ spec status change,
		0)
	~	Dmnl
	~	This level variable is used to specify the status of the requirements \
		specification development/rework status. It can have 3 values: 0,1,2. 0 \
		means it is non-existant. 1 means it is active. 2 means it is active.
	|

Requ spec ver status= INTEG (
	Requ spec ver change,
		0)
	~	Dmnl
	~	This level variable is used to specify the status of the requirements \
		specification activity. It can have 4 values: 0,1,2,3. 0 means that it is \
		non-existant. 1 means that it is active. 2 means it is complete but it has \
		to be repeated. 3 means it is complete.
	|

Sum requ spec faults pending=
	SUM(Requ spec faults pending[origin!,factor!])
	~	Defect
	~	This variable is used to specify the sum of pending requirements \
		specification defects over all origins and factors.
	|
******************************************************
	.Requirement Specification Workforce
******************************************************~
	|

Actual requ dev effort rate=
	IF THEN ELSE( Requ spec learning status<=1,Requ dev workforce,0)
	~	Person
	~	This rate is used to specify the number of developers which are assigned \
		to requirements specification development activity
	|

Actual requ spec effort=
	Actual requ dev effort+Sum actual requ spec rework effort
	~	Person*Day
	~	This variable specifies the amount of actual effort spent on requirements \
		specification development/rework.
	|

Sum actual requ spec rework effort=
	SUM(Actual requ spec rework effort[origin!,factor!])
	~	Person*Day
	~	This variable is used to sum the actual number effort spent on fixing \
		defects detected in the requirements specification document over all \
		origins and quality factors.
	|

Actual requ spec rework effort[origin,factor]= INTEG (
	Actual requ spec rework effort rate[origin,factor],
		0)
	~	Person*Day
	~	This level is used to store the actual rework effort which has been spent \
		on fixing defects detected in the requirements specification document.
	|

Actual requ spec rework effort rate[origin,factor]=
	Requ spec rework effort per fault*Requ spec fault correction rate[origin,factor]
	~	Person
	~	This rate is used to specify the actual rework effort that is spent on \
		fixing defects detected in the requirements specification everyday. It is \
		defined because sometimes developers are assigned to rework some part of \
		the document within a whole day, but the actual work requires less effort. \
		This happens because time step is defined as a day and development \
		assignments are done everyday.
	|

Actual requ dev effort= INTEG (
	Actual requ dev effort rate,
		0)
	~	Day*Person
	~	This level variable is used to store the amount of effort spent of \
		requirements specification development activity.
	|

Initial max requ spec rate per person and day=
	0.07
	~	Page/(Person*Day)
	~	This constant specifies the number of pages that developer initially \
		develop the requirements specification document everyday.
	|

Requ dev effort= INTEG (
	Requ dev workforce,
		0)
	~	Person*Day
	~	This level variable is used to keep track of the effort spent of the \
                requirements specification development/rework acitivity.
    |

Requ ver effort= INTEG (
        Requ ver workforce,
                0)
    ~           Person*Day
    ~           This level variable is used to keep track of the effort spent of the \
                requirements specification verification acitivity.
    |

Requ effort=
        Requ dev effort+Requ ver effort
    ~           Day*Person
    ~           This variable is used to show the total effort spent on the requirements \
                specification document.
    |

Requ ver team skill level average=
        Actual allocation[REV,SKLL]
    ~           Dmnl
    ~           This variable is used to specify the skill level average of the developers \
                assigned to requirements specification verification activity on a 0 to 1 \
                basis.
    |

Requ ver workforce=
        Actual allocation[REV,NMBR]
    ~           Person
    ~           This variable is used to specify the number of developers which are \
                assigned to requirements specification verification activity.
    |

Requ dev team skill level average=
	Actual allocation[RED,SKLL]
	~	Dmnl
	~	This variable is used to specify the skill level average of the developers \
		assigned to requirements specification development/rework activity on a 0 \
		to 1 basis.
	|

Requ dev workforce=
	Actual allocation[RED,NMBR]
	~	Person
	~	This variable is used to specify the number of developers that are \
		assigned to requirements specification development/rework activity.
	|

********************************************************
	.Design Process
********************************************************~
	|

Maximum design dev rate per day[subsystem]=
	IF THEN ELSE(Design learning status[subsystem]>1:AND:Sum design faults pending per subsystem\
		[subsystem]>0:AND:Design to do size[subsystem]>0,Design to do size[subsystem]*Sum design fault correction per subsystem\
		[subsystem]/Sum design faults pending per subsystem[subsystem],Initial design dev rate per person per day\
		[subsystem]*Design dev workforce per subsystem[subsystem])
	~	Page/Day
	~		|

Design dev productivity[subsystem]=
	Maximum design dev rate per day[subsystem]*Design dev team skill level average
	~	Page/Day
	~		|
```

Design ver productivity=

IF THEN ELSE(Number of subsystems being verified in design>0,(Design ver workforce/Number of documents being processed per activity

[DEV]

)*Maximum design ver rate per person per day*Design ver team skill level average,0)

~ Page/Day

~ |

Design to rework due to verification=

SUM(Design to rework due to verification or not[subsystem!])

~ Page/Day

~ This variable sums the 'Design to rework due to verification or not' \

overall subsystems.

|

Design to rework due to verification or not[subsystem]=

IF THEN ELSE(Design to rework[subsystem]>0:AND:Design verification activity[subsystem\

]>0,Design to rework[subsystem],0)

~ Page/Day

~ This variable specifies per subsystem the amount of design rework which is \

being done due to defects detected during verification activity.

|

Design to CM[subsystem]=

IF THEN ELSE(Design doc ready size[subsystem]>Average design size in pages[subsystem\

]-0.1:AND:Total code doc dev status per subsystem

[subsystem]=0,Design doc ready size[subsystem]/TIME STEP,IF THEN ELSE(Total code doc dev status per subsystem\

[subsystem]>0,Design doc ready size

[subsystem]/TIME STEP,0))

~ Page/Day

~ |

Design verification activity[subsystem]=

IF THEN ELSE(Design doc ver status[subsystem]=1:OR:(Design doc ver status[subsystem]\

```
		=2:AND:Design doc dev status[subsystem
	]=2),MIN(Design doc size[subsystem]/TIME STEP,Design ver productivity),0)
	~	Page/Day
	~		|

Design not to verify[subsystem]=
	IF THEN ELSE(Design doc ver status[subsystem]>=3,Design doc size[subsystem]/TIME STEP\
		,IF THEN ELSE(Design doc dev status[subsystem
	]=2:AND:Design doc ver status[subsystem]=3, Design doc size[subsystem]/TIME STEP,IF THEN ELSE\
		(Verify design or not=0,Design doc size
	[subsystem]/TIME STEP,0)))
	~	Page/Day
	~		|

Design development activity[subsystem]=
	IF THEN ELSE(Design doc dev status[subsystem]=1,MIN(Design to do size[subsystem]/TIME STEP\
		,Design dev productivity[subsystem]),0)
	~	Page/Day
	~		|

Design to rework[subsystem]=
	IF THEN ELSE(Design doc quality ratio[subsystem]>0,Design verification activity[subsystem\
		]*MIN(1,Design doc quality ratio
	[subsystem])+Design doc stored flush[subsystem]+Design doc stored late flush[subsystem\
		],Design verification activity[subsystem]+Design doc stored flush[subsystem]+Design doc stored late flush\
		[subsystem])
	~	Page/Day
	~		|

Design doc stored size[subsystem]= INTEG (
	Design to CM[subsystem]-Design doc stored flush[subsystem]-Design doc stored late flush\
		[subsystem],
		0)
	~	Page
	~		|

Design to develop[subsystem]=
	IF THEN ELSE(Requ to CM>0:AND:Total design doc dev status=0, Average design size in pages\
		[subsystem]/TIME STEP, 0)
	~	Page/Day
	~		|

Design doc quality ratio[subsystem]=
	IF THEN ELSE(Design verification activity[subsystem]>0:AND:Design doc quality limit per size unit\
		>0, Sum design fault detection
	[subsystem]/(Design doc quality limit per size unit*Design verification activity[subsystem\
		]), 0)
	~	Dmnl
	~		|

Number of subsystems per product=
	ELMCOUNT(subsystem)
	~	Dmnl
	~	This constant specifies the number of subsystems existing within the \
		system.
	|

Average design size in pages[subsystem]=
	(Average requ size in pages/Number of subsystems per product)*Average requ to design conversion factor per subsystem\
		[subsystem]
	~	Page
	~	This variable specifies the size of the design document for every \
		subsystem. It assumes that each subsystem corresponds to equal number of \
		pages of the requirements specification document.
	|

Actual design size to develop[subsystem]= INTEG (
	Design to develop[subsystem],
		0)
	~	Page
	~		|

Average requ to design conversion factor per subsystem[subsystem]=
	30,28,35,22,40
	~	Dmnl
	~	This constant specifies each page of the requirements specification \
		document convert to how many pages of design document for every subsystem.
	|

Design doc ready size[subsystem]= INTEG (
	Design not to rework[subsystem]+Design not to verify[subsystem]-Design to CM[subsystem\
		],
		0)
	~	Page
	~		|

Design doc size[subsystem]= INTEG (
	Design development activity[subsystem]-Design verification activity[subsystem]-Design not to verify\
		[subsystem],
		0)
	~	Page
	~		|

Design not to rework[subsystem]=
	MAX(Design verification activity[subsystem]-Design to rework[subsystem],0)
	~	Page/Day
	~		|
Design to do size[subsystem]= INTEG (
	Design to develop[subsystem]-Design development activity[subsystem]+Design to rework\
		[subsystem],
		0)
	~	Page
	~		|

Productivity design learning amplifier=
	1
	~	Dmnl
	~		|

********************************************************
	.Design Quality
********************************************************~
	|

Design ver effectiveness constant=
	0.76
	~	Dmnl
	~		|

Average design ver effectiveness[origin,factor]=
	Design ver effectiveness[origin,factor]*Design ver team skill level average
	~	Dmnl
	~		|

Detected design faults flush[origin,factor,subsystem]=
	IF THEN ELSE( Design doc ver status[subsystem]>1, Design faults detected[origin,factor\
		,subsystem]/TIME STEP, 0)
	~	Defect/Day
	~		|
Corrected design faults flush[origin,factor,subsystem]=
        IF THEN ELSE( Design doc dev status[subsystem]=2 , Design faults corrected[origin,factor\
                ,subsystem]/TIME STEP, 0)
        ~       Defect/Day
        ~               |

Requ to design faults waiting[origin,factor,subsystem]= INTEG (
        Requ to design fault propagation[origin,factor]-Design fault generation due to propagation\
                [origin,factor,subsystem],
                0)
        ~       Defect
        ~       This level variable is used to hold the design faults which will be \
                injected in the design document due to defect propagation from the \
                requirements specification phase.
        |

Requ to design fault propagation[origin,factor]=
        IF THEN ELSE(Requ to CM>0:AND:Total design doc dev status=0,Requ spec faults undetected\
                [origin,factor]*Average requ to design fault multiplier
        [origin]/Number of subsystems per product/TIME STEP,0)
        ~       Defect/Day
        ~       This variable specifies the number of defects which will be injected in \
                the design document because of faults in the requirements specification \
                document.
        |

Design ver effectiveness[requ,factor]=
        Design ver effectiveness constant ~~|
Design ver effectiveness[design,factor]=
        Design ver effectiveness constant ~~|
Design ver effectiveness[code,factor]=
        0,0,0
        ~       Dmnl
        ~               |
Design fault detection[requ,factor,subsystem]=
        IF THEN ELSE( Design verification activity[subsystem]>0, MIN(Design faults undetected\
                [requ,factor,subsystem]/TIME STEP
        ,Average design ver effectiveness[requ,factor]*Design verification activity[subsystem\
                ]*(Design faults undetected[requ,factor
        ,subsystem]+Design faults detected[requ,factor,subsystem])/(Design doc size[subsystem\
                ]+Design doc verified[subsystem]))
        ,Code fault detection per subsystem[requ,factor,subsystem]/3) ~~|
Design fault detection[design,factor,subsystem]=
        IF THEN ELSE( Design verification activity[subsystem]>0, MIN(Design faults undetected\
                [design,factor,subsystem]/TIME STEP
        ,Average design ver effectiveness[design,factor]*Design verification activity[subsystem\
                ]*(Design faults undetected
        [design,factor,subsystem]+Design faults detected[design,factor,subsystem]
        )/(Design doc size[subsystem]+Design doc verified[subsystem])),Code fault detection per
subsystem\
                [design,factor,subsystem]/3) ~~|
Design fault detection[code,factor,subsystem]=
        0
        ~       Defect/Day
        ~               |

Design fault correction[origin,factor,subsystem]=
        IF THEN ELSE(Design dev workforce per subsystem[subsystem]>0:AND:Design doc dev status\
                [subsystem]<3, MIN(Design dev workforce per subsystem[subsystem]/Design rework
effort per fault\
                [subsystem],Design faults pending[origin,factor,subsystem]/TIME STEP),IF THEN
ELSE(\
                Design dev workforce per subsystem[subsystem]>0:AND:Design doc dev
status[subsystem\
                ]>=3, MIN(Design dev workforce per subsystem[subsystem]/Design rework effort per
fault\
                [subsystem],Design faults pending[origin,factor,subsystem]/TIME STEP)*Rework design
or not\
                ,0))
```

~ Defect/Day

~ |

Design fault generation[origin,factor,subsystem]=

Design fault generation due to propagation[origin,factor,subsystem]+Design development activity\

[subsystem]*Average design fault injection per size unit

[origin,factor]*(1/MAX(1,Design learning status[subsystem]^Learning amplifier for design fault injection\

))

~ Defect/Day

~ |

Average design fault injection per size unit[origin,factor]=

Minimum design fault injection per size unit[origin,factor]+(1-Design dev team skill level average\

)*Minimum design fault injection per size unit[origin,factor]

~ Defect/Page

~ |

Requ to design faults propagated[origin,factor,subsystem]= INTEG (

Design fault generation due to propagation[origin,factor,subsystem],

0)

~ Defect

~ This level variable specifies the number of faults which were injected in \

the design document because of defects propagated from the requirements \

specification phase.

|

Design fault generation due to propagation[origin,factor,subsystem]=

IF THEN ELSE(Actual design size to develop[subsystem]>0:AND:Requ to design faults waiting\

[origin,factor,subsystem]>0, MIN(Requ to design faults waiting[origin,factor,subsystem\

]/TIME STEP, (Requ to design faults waiting[origin,factor,subsystem]+Requ to design faults propagated\

```
		[origin,factor,subsystem])*Design development activity[subsystem]/Actual design size to develop\
		[subsystem]), 0)
	~	Defect/Day
	~	This rate is used to inject design faults due to defect propagation from \
		the requirements specification phase in the design document as design \
		development happens. It specifies the number of design faults which will \
		be injected in the design document because of propagated requirements \
		specification defects everyday.
	|

Average requ to design fault multiplier[origin]=
	3,0,0
	~	Dmnl
	~	This constant specifies the number of design faults which will be injected \
		in the design document, because of one fault in the requirements \
		specification document.
	|

Sum design fault detection[subsystem]=
	SUM(Design fault detection[origin!,factor!,subsystem])
	~	Defect/Day
	~		|

Actual design faults detected[origin,factor,subsystem]= INTEG (
	Design fault detection[origin,factor,subsystem],
		0)
	~	Defect
	~		|

Design faults detected[origin,factor,subsystem]= INTEG (
	Design fault detection[origin,factor,subsystem]-Detected design faults flush[origin,\
		factor,subsystem],
		0)
	~	Defect
	~		|

Design faults generated[origin,factor,subsystem]= INTEG (
	Design fault generation[origin,factor,subsystem],
		0)
	~	Defect
	~		|

Minimum design fault injection per size unit[requ,factor]=
	0,0,0 ~~|
Minimum design fault injection per size unit[design,factor]=
	0.454,0.454,0.454 ~~|
Minimum design fault injection per size unit[code,factor]=
	0,0,0
	~	Defect/Page
	~		|

Actual design faults corrected[origin,factor,subsystem]= INTEG (
	Design fault correction[origin,factor,subsystem],
		0)
	~	Defect
	~		|

Design faults corrected[origin,factor,subsystem]= INTEG (
	Design fault correction[origin,factor,subsystem]-Corrected design faults flush[origin\
		,factor,subsystem],
		0)
	~	Defect
	~		|

Design faults pending[origin,factor,subsystem]= INTEG (
	Design fault detection[origin,factor,subsystem]-Design fault correction[origin,factor\
		,subsystem],
		0)
	~	Defect
	~		|

Design faults undetected[origin,factor,subsystem]= INTEG (
	Design fault generation copy[origin,factor,subsystem]-Design fault detection[origin,\
		factor,subsystem],
		0)
	~	Defect
	~		|

Design fault generation copy[origin,factor,subsystem]=
	Design fault generation[origin,factor,subsystem]
	~	Defect/Day
	~		|

Learning amplifier for design fault detection=
	2
	~	Dmnl
	~		|

********************************************************
	.Design Status
********************************************************~
	|

Design learning status change[subsystem]=
	IF THEN ELSE(Actual design size to develop[subsystem]>0, (Design development activity\
		[subsystem]+Design verification activity[subsystem])/Actual design size to develop
	[subsystem], 0)
	~	Dmnl/Day
	~		|

Design doc ver status change[subsystem]=
	IF THEN ELSE(Design doc ver status[subsystem]=0:AND:Design doc dev status[subsystem]\
		>1:AND:Verify design or not=1,1,IF THEN ELSE
	(Design doc ver status[subsystem]=1:AND:Design doc size[subsystem]<=0:AND:Design doc quality flag\
		[subsystem]>0:AND:Verify design or not=1,1,IF THEN ELSE(Design doc ver status[subsystem\
		]=2:AND:Design doc size[subsystem]>0:AND:Design doc dev status[subsystem]<>1:AND:Verify design or not\
		=1,-1,IF THEN ELSE(Design doc ver status[subsystem]=1:AND:Design doc size[subsystem\
		]<=0:AND:Design doc quality flag[subsystem]<1:AND:Verify design or not=1,2,IF THEN ELSE\
		(Design doc ver status[subsystem]=0:AND:Verify design or not=0:AND:Design doc dev status\
		[subsystem]>0,0,0)))))
	~	Dmnl/Day
	~		|

Design doc quality flag[subsystem]=
	IF THEN ELSE(Design doc quality limit per size unit>0:AND:Design doc quality[subsystem\
		]>Design doc quality limit per size unit,1,IF THEN ELSE(Design doc quality limit per size unit\
		=0,0,0))
	~	Dmnl
	~		|

Design learning status[subsystem]= INTEG (
	Design learning status change[subsystem],
		0)
	~	Dmnl
	~		|

Design doc dev status change[subsystem]=
	IF THEN ELSE((Design doc dev status[subsystem]=0):AND:(Design to do size[subsystem]>\
		0),1,IF THEN ELSE
	((Design doc dev status[subsystem]=1):AND:
```

```
	(Design to do size[subsystem]<=0),1,IF THEN ELSE((Design doc dev status[subsystem]=2\
		):AND:(Design to do size[subsystem]>0):AND:(Design doc ver status[subsystem]<>1),-
1\
		,0)))
	~	Dmnl/Day
	~		|

Design doc ver status[subsystem]= INTEG (
	Design doc ver status change[subsystem],
		0)
	~	Dmnl
	~	status 0 : non_exist
		status 1: incomplete
		status 2: complete_repeat
		status 3: complete_final
	|

Design doc dev status[subsystem]= INTEG (
	Design doc dev status change[subsystem],
		0)
	~	Dmnl
	~	status 0 : non_exist
		status 1: incomplete
		status 2: complete
	|

Design doc quality limit per size unit=
	0
	~	Defect/Page
	~		|

********************************************************
	.Design Workforce
********************************************************~
```

```
	|

Actual design rework effort rate[origin,factor,subsystem]=
	Design fault correction[origin,factor,subsystem]*Design rework effort per fault[subsystem\
		]
	~	Person
	~		|

Actual design dev effort[subsystem]= INTEG (
	Actual design dev effort rate[subsystem],
		0)
	~	Day*Person
	~		|

Actual design dev effort rate[subsystem]=
	IF THEN ELSE(Design learning status[subsystem]<1,Design dev workforce per subsystem[\
		subsystem],0)
	~	Person
	~		|

Actual design effort=
	Sum actual design dev effort+Sum actual design rework effort
	~	Day*Person
	~	This variable specifies the amount of actual effort spent on design \
		development/rework activities.
	|

Actual design rework effort[origin,factor,subsystem]= INTEG (
	Actual design rework effort rate[origin,factor,subsystem],
		0)
	~	Day*Person
	~		|

Sum actual design dev effort=
	SUM(Actual design dev effort[subsystem!])
	~	Day*Person
	~	This variable sums the actual effort spent on developement of all \
		subsystems.
	|

Sum actual design rework effort=
	SUM(Actual design rework effort[origin!,factor!,subsystem!])
	~	Day*Person
	~		|

Design dev workforce per subsystem[subsystem]=
	IF THEN ELSE(Number of documents being processed per activity[DED]>0:AND:Design doc dev status\
		[subsystem]=1,Design dev workforce/Number of documents being processed per activity
	[DED],0)
	~	Person
	~	This variable specifies how many developers are working on every subsystem.
	|

Design dev effort= INTEG (
	Design dev effort rate,
		0)
	~	Day*Person
	~		|

Design dev effort rate=
	Design dev workforce
	~	Person
	~		|

Design ver effort= INTEG (
	Design ver effort rate,
		0)
	~	Day*Person
	~	This level variable is used to keep track of the effort spent of the \
		design verification acitivity.
	|

Design ver effort rate=
	Design ver workforce
	~	Person
	~		|

Initial design dev rate per person per day[subsystem]=
	0.8287
	~	Page/(Day*Person)
	~		|

Design effort=
	Design dev effort+Design ver effort
	~	Day*Person
	~	This variable is used to show the total effort spent on the design \
		document.
	|

Design dev team skill level average=
	Actual allocation[DED,SKLL]
	~	Dmnl
	~		|

Design ver workforce=
	Actual allocation[DEV,NMBR]
	~	Person
	~		|

Design ver team skill level average=
	Actual allocation[DEV,SKLL]
	~	Dmnl
	~		|

Maximum design ver rate per person per day=
	30
	~	Page/(Day*Person)
	~		|

Design dev workforce=
	Actual allocation[DED,NMBR]
	~	Person
	~		|

********************************************************
	.IT Workforce
********************************************************~
	|

IT effort per module[module]=
	IF THEN ELSE(Number of modules per subsystem[subsystem]>0:AND:Sum code to be tested in IT per subsystem\
		[subsystem]>
	0,IT effort per subsystem[subsystem]*(Code to be tested in IT[module]/Sum code to be tested in IT per subsystem\
		[subsystem]),0)
	~	Day*Person
	~	This variable specifies the amount of effort that has to be spent for \
		integration testing for a module.
	~	:SUPPLEMENTARY
	|

Integration testing effort=
	SUM(IT execution effort per module[module!])+SUM(Integration testing TC dev effort per subsystem\
		[subsystem!])
	~	Day*Person
	~	This variable is used to show the total effort spent for integration \
		testing.
	|

IT execution effort rate per module[module]=
	IF THEN ELSE(IT rate[module]>0,IT workforce per module[module],0)
	~	Person
	~	This rate specifies the number of developers that work on integration \
		testing execution of a module everyday.
	|

IT execution effort per module[module]= INTEG (
	IT execution effort rate per module[module],
		0)
	~	Day*Person
	~	This level variable specifies the amount of effort that is spent on \
		integration testing execution activity for a module.
	|

Integration testing effort rate per subsystem=
	Integration testing execution workforce
	~	Person
	~		|

IT workforce per module[module]=
	IF THEN ELSE(Number of documents being processed per activity[ITV]>0,Integration testing execution workforce\
		/(Number of documents being processed per activity[ITV]*Number of modules per subsystem\
		[subsystem]),0)
	~	Person
	~		|

IT effort per subsystem[subsystem]=
```

Sum code to be tested in IT per subsystem[subsystem]/Average IT productivity per person per day

~ Day*Person

~ This variable specifies the amount of effort that has to be spent for \

integration testing for a subsystem.

|

Integration testing execution effort per subsystem= INTEG (

Integration testing effort rate per subsystem,

0)

~ Day*Person

~ |

Integration testing TC dev effort per subsystem[subsystem]= INTEG (

Integration testing TC dev effort rate per subsystem[subsystem],

0)

~ Day*Person

~ |

Integration testing TC dev effort rate per subsystem[subsystem]=

IF THEN ELSE(Subsystem under TC dev in IT[subsystem]>0,Integration testing TC dev workforce\

/Number of subsystems under TC dev in IT,0)

~ Person

~ |

Integration testing execution team skill level average=

Actual allocation[ITV,SKLL]

~ Dmnl

~ |

Integration testing TC dev team skill level average=

Actual allocation[ITTC,SKLL]

~ Dmnl

~ |

```
Integration testing TC dev workforce=
        Actual allocation[ITTC,NMBR]
        ~       Person
        ~               |

Integration testing execution workforce=
        Actual allocation[ITV,NMBR]
        ~       Person
        ~               |

Maximum number of IT test cases developed per person per day=
        5
        ~       Testcase/(Day*Person)
        ~               |

Average number of IT test cases executed per person per day=
        20
        ~       Testcase/(Day*Person)
        ~               |

********************************************************
        .ST Workforce
********************************************************~
        |

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 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 test cases executed per person per day=
	20
	~	Testcase/(Day*Person)
	~		|
********************************************************
	.Code Process
********************************************************~
	|

Maximum code dev rate per day[module]=
	IF THEN ELSE(Code learning status[module]>=1:AND:Sum code faults pending per module[\
		module]>0,Code to do size[module]*Sum code fault correction per module
	[module]/Sum code faults pending per module[module], Initial code dev rate per person per day\
		*Code dev workforce per module
	[module])
	~	KLOC/Day
	~		|

Code dev productivity[module]=
	Maximum code dev rate per day[module]*Code dev team skill level average
	~	KLOC/Day
	~		|

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
	~		|

Portion of code reworked to actual system code size per subsystem[subsystem]=
	Portion of code reworked to actual subsystem code size per subsystem[subsystem]/Number of subsystems per product\
		/TIME STEP
	~	Dmnl/Day
	~	This variable is used to specify the portion of code reworked to actual \
		system code size per subsystem
	|

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 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 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
	~		|

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
	~		|

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 Quality
********************************************************~
```

|

Code ver effectiveness constant=

0.53

~ Dmnl

~ |

Average code ver effectiveness[origin,factor]=

Code ver effectiveness[origin,factor]*Code ver team skill level average

~ Dmnl

~ |

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 Status
********************************************************~
	|

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 Workforce
********************************************************~
	|

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[origin,factor,module]=
	Code fault correction[origin,factor,module]*Code rework effort per fault[module]
	~	Person
	~		|

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 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
	~		|

Maximum code ver rate per person per day=
	0.6
	~	KLOC/(Day*Person)
	~		|

Code dev workforce=
	Actual allocation[COD,NMBR]
	~	Person
	~		|

********************************************************
	.IT Process
********************************************************~
	|

Average IT productivity[module]=
	IF THEN ELSE(IT effort per module[module]>0,Code to be tested in IT[module]*IT workforce per module\
		[module]/IT effort per module[module],0)
	~	KLOC/Day
	~	This rate specifies the number of KLOCs of code document that is \
		integration tested everyday.
	|

Average IT productivity per person per day=
	0.1856
	~	KLOC/(Person*Day)
	~	This constant specifies the number of KLOCs of code document that each \
		developer integration tests everyday
	|

Average number of IT test cases developed per day=
	Maximum number of IT test cases developed per person per day*Integration testing TC dev workforce\
		*Integration testing TC dev team skill level average
	~	Testcase/Day
	~		|

IT test case dev rate[subsystem]=
	IF THEN ELSE(Subsystem waiting for TC dev for IT[subsystem]=1:AND:Average number of IT test cases developed per day\
		>0:AND:
	IT test cases[subsystem]<Number of test cases for IT
	[subsystem]:AND:IT test cases[subsystem]/TIME STEP+Average number of IT test cases developed per day\
		<=Number of test cases for IT
	[subsystem]/TIME STEP,Average number of IT test cases developed per day,IF THEN ELSE\
		(Subsystem waiting for TC dev for IT[subsystem
	]:AND:Average number of IT test cases developed per day
	>0:AND:IT test cases[subsystem]<Number of test cases for IT[subsystem]:AND:IT test cases\
		[subsystem]/TIME STEP+Average number of IT test cases developed per day
	>Number of test cases for IT[subsystem]/TIME STEP,(Number of test cases for IT[subsystem\
		]-IT test cases[subsystem])/TIME STEP,0))
	~	Testcase/Day
	~		|

Code returned for rework from IT[module]=
	IF THEN ELSE(Integration test status[module]>1,Tested code in IT[module]/TIME STEP,0\
		)
	~	KLOC/Day
	~		|

Code ready for IT flush[module]=
	IF THEN ELSE(Code ready for IT for a module's subsystem[module]<(Actual code size of a module's subsystem\
		[module]+0.1):AND:
	Code ready for IT for a module's subsystem[module]>(Actual code size of a module's subsystem\
		[module]-0.1),Code ready for IT
	[module]/TIME STEP,0)
	~	KLOC/Day
	~	This rate is used to send all modules of a subsystem for integration \
		testing to begin. It does that when all modules of a subsystem are ready \
		for integration testing.
	|

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

Incoming code to IT[module]=
	Code to IT flush[module]
	~	KLOC/Day
	~		|

Sum code ready for IT per subsystem[subsystem]=
	CUSTOMSUMONED(Code ready for IT[MOD1], Subsystem's first module number[subsystem],Number of modules per subsystem\
		[subsystem])
	~	KLOC
	~	This variable specifies the amount of subsystem's code that is ready for \
		integration testing
	|
Average number of IT test cases executed per day=
        Average number of IT test cases executed per person per day*Integration testing execution workforce
        ~       Testcase/Day
        ~               |

Code ready for IT for a module's subsystem[module]=
        Sum code ready for IT per subsystem[subsystem]+Sum code ready for ST per subsystem[subsystem\
                ]
        ~       KLOC
        ~       This variable specifies the number of KLOCs of code that is ready for \
                integration testing for a module's subsystem. for example, for the first \
                module its the amount of code from the first subsystem which is ready for \
                integration testing.'Sum code ready for ST per subsystem' is added because \
                even when some of a subsystem's code passes integration testing it is \
                needed for integration testing of the rest on the code of the subsystem.
        |

Incoming code ready for IT rate[module]=
        Code ready for IT flush[module]
        ~       KLOC/Day
        ~               |

Code to be tested in IT[module]= INTEG (
        Incoming code ready for IT rate[module]-IT rate[module],
                0)
        ~       KLOC
        ~               |

Actual code size of a module's subsystem[module]=
        Sum actual code size to develop per subsystem[subsystem]
        ~       KLOC
        ~       This variable specifies the code size of a module's subsystem (the \
                subsystem that the module belongs to).
```

```
	|

Code ready for IT[module]= INTEG (
	Incoming code to IT[module]-Code ready for IT flush[module],
		0)
	~	KLOC
	~	This level variable specifies the amount of a module's code that is ready \
		for integration testing.
	|

Tested code in IT[module]= INTEG (
	IT rate[module]-Code returned for rework from IT[module],
		0)
	~	KLOC
	~		|

********************************************************
	.IT Quality
********************************************************~
	|

Integration testing TC dev team skill level average per subsystem[subsystem]=
	IF THEN ELSE(Subsystem under TC dev in IT[subsystem]=1,Integration testing TC dev team skill level average\
		/TIME STEP,0)
	~	Dmnl/Day
	~		|

IT TC dev team skill level average average[subsystem]=
	IF THEN ELSE(IT TC dev done or not[subsystem]=1,IT TC dev team skill level average stored\
		[subsystem]/IT TC dev team working time
	[subsystem]*TIME STEP,0)
	~	Dmnl
	~		|
Average IT effectiveness[origin,factor,subsystem]=
	IF THEN ELSE(IT test case data available or not=1,IT effectiveness[origin,factor]*IT TC dev team skill level average average
	[subsystem],IT effectiveness[origin,factor]*Integration testing execution team skill level average\
		)
	~	Dmnl
	~		|

IT TC dev team skill level average stored[subsystem]= INTEG (
	IT TC dev team skill level average rate[subsystem],
		0)
	~	Dmnl
	~		|

IT TC dev team working time[subsystem]= INTEG (
	IT TC dev team working rate[subsystem],
		0)
	~	Day
	~		|

IT TC dev team skill level average rate[subsystem]=
	Integration testing TC dev team skill level average per subsystem[subsystem]
	~	Dmnl/Day
	~		|

IT TC dev team working rate[subsystem]=
	IF THEN ELSE(Integration testing TC dev team skill level average per subsystem[subsystem\
		]>0,1,0)
	~	Dmnl
	~		|

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

Undetected code faults in IT flush[origin,factor,module]=
    IF THEN ELSE(Integration test status[module]>1,Undetected code faults in IT[origin,factor\
        ,module]/TIME STEP,0)
    ~       Defect/Day
    ~               |

Incoming code faults to IT rate[origin,factor,module]=
    IF THEN ELSE(Code ready for IT flush[module]>0,Code faults undetected in coding[origin\
        ,factor,module]/TIME STEP,0)
    ~       Defect/Day
    ~               |

Detected code faults in IT flush[origin,factor,module]=
    IF THEN ELSE(Integration test status[module]>1,Detected code faults in IT[origin,factor\
        ,module]/TIME STEP,0)
    ~       Defect/Day
    ~               |

Detected code faults in IT[origin,factor,module]= INTEG (
    Code fault detection rate in IT[origin,factor,module]-Detected code faults in IT flush\
        [origin,factor,module],
        0)
    ~       Defect
    ~               |
Undetected code faults in IT[origin,factor,module]= INTEG (
        Incoming code faults to IT rate[origin,factor,module]-Code fault detection rate in IT\
                [origin,factor,module]-Undetected code faults in IT flush[origin,factor,module],
                0)
        ~       Defect
        ~               |

Actual code faults detected in IT[origin,factor,module]= INTEG (
        Actual code faults detected in IT rate[origin,factor,module],
                0)
        ~       Defect
        ~               |

Actual code faults detected in IT rate[origin,factor,module]=
        Code fault detection rate in IT[origin,factor,module]
        ~       Defect/Day
        ~               |

IT effectiveness[requ,factor]=
        0.69,0.69,0.69 ~~|
IT effectiveness[design,factor]=
        0.69,0.69,0.69 ~~|
IT effectiveness[code,factor]=
        0.69,0.69,0.69
        ~       Dmnl
        ~               |

********************************************************
        .IT Status
********************************************************~
        |

Ready for IT exec flag for subsystems[subsystem]=
	IF THEN ELSE(CUSTOMSUMONED(Ready for IT exec flag for modules[MOD1],Subsystem's first module number\
		[subsystem],Number of modules per subsystem[subsystem])>0,1,0)
	~	Dmnl
	~	This variable specifies if a subsystem is ready for integration testing \
		execution.
	|

Code ready for IT TC dev flag for subsystems[subsystem]=
	IF THEN ELSE(CUSTOMSUMONED(Code ready for IT TC dev flag for modules[MOD1],Subsystem's first module number\
		[subsystem],Number of modules per subsystem[subsystem])>0,1,0)
	~	Dmnl
	~	This variable specifies if a subsystem is ready for integration testing \
		test case development.
	|

Number of modules being integration tested=
	SUM(Ready for IT exec flag for modules[module!])
	~	Dmnl
	~		|

Module under IT execution[module]=
	IF THEN ELSE(Integration test status[module]=1,1,0)
	~	Dmnl
	~	This variable specifies if a module is under integration testing execution.
	|

Subsystem under IT execution[subsystem]=
	IF THEN ELSE(Sum IT rate per subsystem[subsystem]>0,1,0)
	~	Dmnl
	~	This variable specifies if a subsystem is under integration testing \
		execution.
	|
```

Number of subsystems being integration tested=

SUM(Subsystem under IT execution[subsystem!])

~ Dmnl

~ This variable specifies the number of subsystems that are under \
integration testing execution.

|

Ready for IT exec flag for modules[module]=

IF THEN ELSE(IT test case data available or not=1:AND:Code to be tested in IT[module\
]>0:AND:IT test cases[subsystem]>Number of test cases for IT[subsystem]-0.001,1,IF THEN ELSE\
(IT test case data available or not=0:AND:Code to be tested in IT[module]>0,1,0))

~ Dmnl

~ This variable specifies if a module is ready for integration testing \
execution.

|

Subsystem waiting for TC dev for IT[subsystem]=

IF THEN ELSE(IT test case data available or not=1,IF THEN ELSE(Design doc stored flag\
[subsystem]=1:AND:IT test cases[subsystem]<Number of test cases for IT[subsystem]-0.0001
:AND:Postpone TC dev till code is ready in IT or not=0,1,IF THEN ELSE(Postpone TC dev till code is ready in IT or not\
=1
:AND:Code ready for IT TC dev flag for subsystems[subsystem]=1:AND:IT test cases[subsystem\
]<Number of test cases for IT[
subsystem]-0.0001,1,0)),0)

~ Dmnl

~ This variable specifies if a subsystem is waiting for integration testing \
test case development.

|

Number of subsystems under TC dev in IT=

SUM(Subsystem under TC dev in IT[subsystem!])

```
	~	Dmnl
	~		|

Subsystem under TC dev in IT[subsystem]=
	IF THEN ELSE(IT test case dev rate[subsystem]>0,1,0)
	~	Dmnl
	~		|

IT TC dev done or not[subsystem]=
	IF THEN ELSE(Number of test cases for IT[subsystem]>0:AND:IT test cases[subsystem]>Number
of test cases for IT\
		[subsystem]-0.001,1,0)
	~	Dmnl
	~		|

Code ready for IT TC dev flag for modules[module]=
	IF THEN ELSE(Code to be tested in IT[module]>0,1,0)
	~	Dmnl
	~	This variable specifies if a module is ready for integration testing test \
		case development.
	|

Postpone TC dev till code is ready in IT or not=
	1
	~	Dmnl
	~	This variable specifies if integration test case development can be done \
		right after design development & verification is finished or we can \
		develop integration test cases only when the code document of the whole \
		subsystem is ready. If set to 1, integration test case development is done \
		after all subsystem code is ready and if set to 0, right after design \
		development & verification is finished.
	|

Design doc stored flag[subsystem]=
	IF THEN ELSE(Design doc stored size[subsystem]>0,1,0)
	~	Dmnl
	~	This variable specifies if a subsystem design is stored in the \
		configuration management.
	|

Number of subsystems waiting for TC dev in IT=
	SUM(Subsystem waiting for TC dev for IT[subsystem!])
	~	Dmnl
	~		|

Number of subsystems waiting for integration testing=
	SUM(Ready for IT exec flag for subsystems[subsystem!])
	~	Dmnl
	~		|

Total integration test status=
	SUM(Integration test status[module!])/(3*SUM(Number of modules per subsystem[subsystem\
		!]))
	~	Dmnl
	~		|

Module quality in IT[module]=
	IF THEN ELSE(IT rate[module]=0:AND:Tested code in IT[module]>0,Sum detected code faults in
IT\
		[module]/Tested code in IT[module],0)
	~	Defect/KLOC
	~		|

Integration test quality threshold=
	0
	~	Defect/KLOC
	~		|
Sum detected code faults in IT[module]=
	SUM(Detected code faults in IT[origin!,factor!,module])
	~	Defect
	~		|

Integration test status change rate[module]=
	IF THEN ELSE(IT rate[module]>0:AND:Integration test status[module]=0,1,IF THEN ELSE(\
		Integration test status[module]=1:AND:IT rate[module]=0:AND:Quality flag in IT[module\
		]=1,1,IF THEN ELSE(Integration test status[module]=1:AND:IT rate[module]=0:AND:Quality flag in IT\
		[module]=0,2,IF THEN ELSE(Integration test status[module]=2:AND:IT rate[module]>0,-\
		1,0))))
	~	Dmnl/Day
	~		|

Integration test status[module]= INTEG (
	Integration test status change rate[module],
		0)
	~	Dmnl
	~		|

Quality flag in IT[module]=
	IF THEN ELSE(Integration test quality threshold>0:AND:Module quality in IT[module]>Integration test quality threshold\
		,1,IF THEN ELSE(Integration test quality threshold=0,0,0))
	~	Dmnl/Day
	~		|

********************************************************
	.Requirement Specificatin Status
********************************************************~
	|
Verify requ spec or not=
        1
        ~       Dmnl
        ~       This switch is used to turn on/off the requirements specification \
                verification activity.
        |

********************************************************
        .UT Status
********************************************************~
        |

Unit test waiting flag[module]=
        IF THEN ELSE(UT Test case data availble or not=1:AND:Code to be tested in UT[module]\
                >0:AND:UT test cases[module]>Number of test cases for UT[module]-0.001,1,IF THEN ELSE\
                (UT Test case data availble or not=0:AND:Code to be tested in UT[module]>0,1,0))
        ~       Dmnl
        ~       This flag specifies if a module is ready for unit test execution.
        |

Module waiting for TC dev for UT[module]=
        IF THEN ELSE(UT Test case data availble or not=1:AND:Code to be tested in UT[module]\
                >0:AND:UT test cases[module]<Number of test cases for UT[module]-0.001,1,0)
        ~       Dmnl
        ~       This flag specifies is a module is waiting for unit test case development \
                or not. If set to 1 it is waiting for unit test case development and if \
                set to 0, it is not.
        |

Sum module under TC dev in UT=
        SUM(Module under TC dev in UT[module!])
        ~       Dmnl
        ~       This variable specifies the sum of 'Module under TC dev in UT' over all \
```

modules.

|

Module under TC dev in UT[module]=

IF THEN ELSE(UT test case dev rate[module]>0,1,0)

~ Dmnl

~ This variable specifies if a module is under unit test case development.

|

UT TC dev done or not[module]=

IF THEN ELSE(Number of test cases for UT[module]>0:AND:UT test cases[module]>Number of test cases for UT\

[module]-0.001,1,0)

~ Dmnl

~ This variable specifies if unit test case development for a module is \
finished or not. If set to 1 it is finished and if set to 0 it is not.

|

Number of modules waiting for TC dev in UT=

SUM(Module waiting for TC dev for UT[module!])

~ Dmnl

~ This variable specifies the number of modules that are waiting for unit \
test case development.

|

Number of modules waiting to be unit tested=

SUM(Unit test waiting flag[module!])

~ Dmnl

~ This variable specifies the number of modules that are waiting for unit \
test execution.

|

Total unit test status=

SUM(Unit test status[module!])/(3*SUM(Number of modules per subsystem[subsystem!]))

```
	~	Dmnl
	~	This variable specifies the status of unit testing of the whole system \
		(all modules) on a 0 to 1 basis.
	|

Unit test status change rate[module]=
	IF THEN ELSE(Unit testing rate[module]>0:AND:Unit test status[module]=0,1,IF THEN ELSE\
		(Unit test status[module]=1:AND:Unit testing rate
	[module]=0:AND:Quality flag in UT[module]=1,1,IF THEN ELSE(Unit test status[module]=\
		1:AND:Unit testing rate[module]=0:AND:Quality flag in UT
	[module]=0,2,IF THEN ELSE(Unit test status[module]=2:AND:Unit testing rate[module]>0\
		,-1,0))))
	~	Dmnl/Day
	~	This rate specifies the change in value of the 'Unit test status' everyday.
	|

Module quality in UT[module]=
	IF THEN ELSE(Unit testing rate[module]=0:AND:Tested code in UT[module]>0,Sum detected code faults in UT per module\
		[module]/Tested code in UT
	[module]
	,0)
	~	Defect/KLOC
	~	This variable specifies the quality of a module code document during the \
		unit testing activity
	|

Quality flag in UT[module]=
	IF THEN ELSE(Unit test quality threshold[module]>0:AND:Module quality in UT[module]>\
		Unit test quality threshold[module],1,IF THEN ELSE(Unit test quality threshold[module\
		]=0,0,0))
	~	Dmnl
	~	This flag shows if the module has an acceptable quality considering the \
		unit test qualit threshold.
	|

Unit test quality threshold[module]=
	0
	~	Defect/KLOC
	~	This variable specifies the acceptable quality for a module after unit \
		test.
	|

Unit test status[module]= INTEG (
	Unit test status change rate[module],
		0)
	~	Dmnl
	~	This level variable shows the status of the unit test activity. \
		status 0 : non_exist
		status 1: incomplete
		status 2: complete_repeat
		status 3: complete_final
	|

********************************************************
	.ST Process
********************************************************~
	|

Average ST productivity=
	System testing execution workforce*Average ST productivity per person per day
	~	KLOC/Day
	~		|

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.
	|
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
	~		|

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 Quality
********************************************************~
	|

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 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)
	~		|

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
********************************************************~
        |

System test flag[module]=
        IF THEN ELSE(ST test case data available or not=1:AND:Code to be tested in ST[module\
                ]>0:AND:ST test cases>Number of test cases for ST-0.0001,1,IF THEN ELSE(ST test case data available or not\
                =0:AND:Code to be tested in ST[module]>0,1,0))
        ~       Dmnl
        ~               |

System waiting for TC dev for ST=
        IF THEN ELSE(ST test case data available or not=1,IF THEN ELSE(Postpone TC dev until code is ready in ST or not\
                =0:AND:Requ doc stored size>0:AND:ST test cases<Number of test cases for ST
```

```
	-0.0001,1,IF THEN ELSE(Postpone TC dev until code is ready in ST or not=1:AND:SUM(Code to be tested in ST\
		[module!])>0:AND:
	ST test cases<Number of test cases for ST
	-0.0001,1,0)),0)
	~	Dmnl
	~		|

System under TC dev in ST=
	IF THEN ELSE(ST test case dev rate>0,1,0)
	~	Dmnl
	~		|

ST TC dev done or not=
	IF THEN ELSE(Number of test cases for ST>0:AND:ST test cases>Number of test cases for ST\
		-0.001,1,0)
	~	Dmnl
	~		|

Postpone TC dev until code is ready in ST or not=
	0
	~	Dmnl
	~	This variable specifies it system test case development can be done right \
		after requirement specification development & verification is finished or \
		we can develop system test cases only when the code document of the whole \
		system is ready. If set to 1, system test case development is done after \
		all system code is ready and if set to 0, right after requirements \
		specification development & verification is finished.
	|

Number of systems waiting to be system tested=
	IF THEN ELSE(SUM(System test flag[module!])>0,1,0)
	~	Dmnl
	~		|
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
	~		|

********************************************************
	.UT Process
********************************************************~
	|

Average UT productivity=
	IF THEN ELSE(Number of documents being processed per activity[UTV],Unit testing TC execution workforce\
		*Average UT productivity per person per day/Number of documents being processed per activity\
		[UTV],0)
	~	KLOC/Day
	~	This variable specifies the number of KLOCs of code document unit tested \
		everyday.
	|

Average number of UT test cases developed per day=
	Unit testing TC dev workforce*Maximum number of UT test cases developed per person per day\
		*Unit testing TC dev skill level average
	~	Testcase/Day
	~	This variable specifies the number of unit test cases that are developed \
		everyday.
	|

Unit testing rate[module]=
	IF THEN ELSE(UT Test case data availble or not=1,IF THEN ELSE(Code to be tested in UT\
		[module]>0:AND:UT test cases[module
```

]>Number of test cases for UT[module]-0.0001:AND:

Average number of UT test cases executed per day>0,MIN(Code to be tested in UT[module\

]/TIME STEP,Actual code size to develop per module

[module]/(Number of test cases for UT[module]/Average number of UT test cases executed per day\

)),0),IF THEN ELSE(Code to be tested in UT

[module]>0,MIN(Code to be tested in UT[module]/TIME STEP,Average UT productivity),0)\

)

~ KLOC/Day

~ This rate specifies the number of KLOCs of code document that are unit \

tested everyday.

|

Code returned for rework rate from UT[module]=

IF THEN ELSE(Unit test status[module]>1,Tested code in UT[module]/TIME STEP,0)

~ KLOC/Day

~ This rate specifies the number of KLOCs of code document that are sent \

back to coding phase for rework.

|

Average number of UT test cases executed per day=

Average number of UT test cases executed per person per day*Unit testing TC execution workforce

~ Testcase/Day

~ This variable specifies the number of unit test cases that are executed \

everyday.

|

Code to unit test rate[module]=

Code to UT flush[module]

~ KLOC/Day

~ This rate variable specifies the number of KLOCs of code document which \

become ready for unit test everyday.

|

```
Tested code in UT[module]= INTEG (
	Unit testing rate[module]-Code returned for rework rate from UT[module],
		0)
	~	KLOC
	~	This level variable specifies the number of KLOCs of code document that \
		are tested during the unit testing activity.
	|

Code to be tested in UT[module]= INTEG (
	Code to unit test rate[module]-Unit testing rate[module],
		0)
	~	KLOC
	~	This level variable specifies the number of KLOCs of code document that \
		are waiting to be unit tested.
	|

********************************************************
	.UT Quality
********************************************************~
	|

UT TC dev team skill level average average[module]=
	IF THEN ELSE(UT TC dev done or not[module]=1,UT TC dev team skill level average stored\
		[module]/UT TC dev team working time[module]*TIME STEP,0)
	~	Dmnl
	~	This variable specifies the average of average skill level of developers \
		who developed unit test cases. This variable is used because within \
		different days different teams of developers with different average skill \
		levels develop the unit test cases.
	|

UT TC dev team skill level average per module[module]=
	IF THEN ELSE(Module under TC dev in UT[module]=1,Unit testing TC dev skill level average\
		/TIME STEP,0)
```

```
        ~       Dmnl/Day
        ~       This variable specifies the average skill level of developers that develop \
                unit test cases everyday.
        |

Average unit testing effectiveness[origin,factor,module]=
        IF THEN ELSE(UT Test case data availble or not=1,Unit testing effectiveness[origin,factor\
                ]*UT TC dev team skill level average average
        [module],Unit testing effectiveness[origin,factor]*Unit testing execution team skill level average\
                )
        ~       Dmnl
        ~       This variable specifies the effectiveness of the unit testing technique in \
                detecting undetected defects in code
        |

UT TC dev team skill level average rate[module]=
        UT TC dev team skill level average per module[module]
        ~       Dmnl/Day
        ~               |

UT TC dev team skill level average stored[module]= INTEG (
        UT TC dev team skill level average rate[module],
                0)
        ~       Dmnl
        ~       This level variable stores the average skill level of unit test case \
                developers that developed the unit test cases.
        |

UT TC dev team working time[module]= INTEG (
        UT TC dev team working rate[module],
                0)
        ~       Day
        ~       This level variable stores the number of days that took to develop unit \
                test cases for a module.
```

```
	|

UT TC dev team working rate[module]=
	IF THEN ELSE(UT TC dev team skill level average per module[module]>0,1,0)
	~	Dmnl
	~		|

Detected code faults in UT flush[origin,factor,module]=
	IF THEN ELSE(Unit test status[module]>1,Detected code faults in UT[origin,factor,module\
		]/TIME STEP,0)
	~	Defect/Day
	~	This rate variable is used to reset the 'Detected code faults in UT' level \
		variable at the end of the unit testing process.
	|

Incoming code faults to UT rate[origin,factor,module]=
	IF THEN ELSE(Code to UT flush[module]>0,Code faults undetected in coding[origin,factor\
		,module]/TIME STEP,0)
	~	Defect/Day
	~	This rate specifies the number of code defects that exist in the code \
		document before unit testing begins.
	|

Code fault detection rate in UT[origin,factor,module]=
	IF THEN ELSE(Unit testing rate[module]>0, MIN(Undetected code faults in UT[origin,factor\
		,module]/TIME STEP, Average unit testing effectiveness
	[origin,factor,module]*Unit testing rate
	[module]*(Undetected code faults in UT[origin,factor,module]+Detected code faults in UT\
		[origin,factor,module])/(Code to be tested in UT
	[module]+Tested code in UT[module])),0)
	~	Defect/Day
	~		|

Undetected code faults in UT flush[origin,factor,module]=
	IF THEN ELSE(Unit test status[module]>1,Undetected code faults in UT[origin,factor,module\
		]/TIME STEP,0)
	~	Defect/Day
	~	This rate variable is used to reset the 'Undetected code faults in UT' \
		level variable at the end of the unit testing process.
	|

Undetected code faults in UT[origin,factor,module]= INTEG (
	Incoming code faults to UT rate[origin,factor,module]-Code fault detection rate in UT\
		[origin,factor,module]-Undetected code faults in UT flush[origin,factor,module],
		0)
	~	Defect
	~	This level variable specifies the number of defects that remain undetected \
		in the code document during the unit test.
	|

Actual code faults detected in UT rate[origin,factor,module]=
	Code fault detection rate in UT[origin,factor,module]
	~	Defect/Day
	~		|

Actual code faults detected in UT[origin,factor,module]= INTEG (
	Actual code faults detected in UT rate[origin,factor,module],
		0)
	~	Defect
	~	This level variable is used to store the number of defects that are \
		detected because of the unit testing activity because the 'Detected code \
		faults in UT' is reset at the end of any unit testing process.
	|

Detected code faults in UT[origin,factor,module]= INTEG (
	Code fault detection rate in UT[origin,factor,module]-Detected code faults in UT flush\
		[origin,factor,module],
		0)
	~	Defect
	~	This rate specifies the number of code defects that are detected in the \
		code document during the unit test process.
	|

Unit testing effectiveness[requ,factor]=
	0.66,0.66,0.66 ~~|
Unit testing effectiveness[design,factor]=
	0.66,0.66,0.66 ~~|
Unit testing effectiveness[code,factor]=
	0.66,0.66,0.66
	~	Dmnl
	~	This constant specifies the maximum effectiveness of the unit testing \
		technique in detecting undetected defects in code. By maximum it is meant \
		that a developer with skill level of 1 will detect this percentage of \
		defects using this unit testing technique.
	|

********************************************************
	.UT Workforce
********************************************************~
	|

Unit test effort=
	Unit test TC dev effort+Unit test TC execution effort
	~	Day*Person
	~	This variable is used to show the total effort spent for unit testing.
	|

Unit test TC execution effort= INTEG (
	Unit testing TC execution workforce,
		0)
	~	Day*Person
	~	This level variable specifies the amount of effort spent on unit test \
		execution.
	|

Unit test TC dev effort= INTEG (
	Unit testing TC dev workforce,
		0)
	~	Day*Person
	~	This level variable specifies the amount of effort spent on developing \
		unit test cases.
	|

Average UT productivity per person per day=
	0.3093
	~	KLOC/(Person*Day)
	~	This constant specifies the number of KLOCs of code document that every \
		developer unit test everyday.
	|

Unit testing TC dev skill level average=
	Actual allocation[UTTC,SKLL]
	~	Dmnl
	~	This variable specifies the skill level average of developers assigned to \
		develop unit test cases.
	|

Unit testing TC dev workforce=
	Actual allocation[UTTC,NMBR]
	~	Person
	~	This variable specifies the number of developers assigned to develop unit \
		test cases.
	|

Unit testing execution team skill level average=
	Actual allocation[UTV,SKLL]
	~	Dmnl
	~	This variable specifies the skill level average of the developers assigned \
		to execute unit test cases.
	|

Maximum number of UT test cases developed per person per day=
	1
	~	Testcase/(Day*Person)
	~	This constant specifies the number of unit test cases that each developer \
		develops everyday.
	|

Average number of UT test cases executed per person per day=
	4
	~	Testcase/(Day*Person)
	~	This constant the number of unit test cases that each developer executes \
		everyday.
	|

Unit testing TC execution workforce=
	Actual allocation[UTV,NMBR]
	~	Person
	~	This variable specifies the number of developers assigned to execute unit \
		test cases.
	|
********************************************************
average number of IT test cases per design size unit=
	5
	~	Testcase/Page
	~	This constant specifies the average number of integration testing test \
		cases that are developed for every page of the design document.
	|

Number of test cases for IT[subsystem]=
	Actual design size to develop[subsystem]*average number of IT test cases per design size unit
	~	Testcase
	~		|

ST test case dev rate=
	IF THEN ELSE(System waiting for TC dev for ST=1:AND:Average number of ST test cases developed per day\
		>0:AND:ST test cases/TIME STEP
	+Average number of ST test cases developed per day<=Number of test cases for ST/TIME STEP\
		,Average number of ST test cases developed per day
	,IF THEN ELSE(System waiting for TC dev for ST=1:AND:Average number of ST test cases developed per day\
		>0:AND:ST test cases/TIME STEP
	+Average number of ST test cases developed per day>Number of test cases for ST/TIME STEP\
		,(Number of test cases for ST-ST test cases)/TIME STEP
	,0))
	~	Testcase/Day
	~		|

UT test case dev rate[module]=
	IF THEN ELSE(Module waiting for TC dev for UT[module]=1:AND:Average number of UT test cases developed per day\
		>0:AND:UT test cases
	[module]<Number of test cases for UT[module]-0.001:AND:UT test cases[module]/TIME STEP\
		+Average number of UT test cases developed per day
	<=Number of test cases for UT[module]/TIME STEP,Average number of UT test cases developed per day\
		,IF THEN ELSE(Module waiting for TC dev for UT
	[module]:AND:Average number of UT test cases developed per day
	>0:AND:UT test cases[module]<Number of test cases for UT[module]-0.001:AND:UT test cases\
		[module]/TIME STEP+Average number of UT test cases developed per day
	>Number of test cases for UT[module]/TIME STEP,(Number of test cases for UT[module]-\
		UT test cases[module])/TIME STEP,0))
	~	Testcase/Day
	~	This rate specifies the number of unit test cases that are developed \
		everyday.
	|

Portion of code reworked to actual subsystem code size per subsystem[subsystem]=
	IF THEN ELSE(CUSTOMSUMONED(Code to rework[MOD1],Subsystem's first module number[subsystem\
		],Number of modules per subsystem
	[subsystem])>0,CUSTOMSUMONED(Code to rework[MOD1],Subsystem's first module number[subsystem\
		],Number of modules per subsystem
	[subsystem])/Sum actual code size to develop per subsystem[subsystem]/TIME STEP,0)
	~	Dmnl
	~	This variable is used to specify the portion of code reworked to actual \
		subsystem code size per subsystem
	|

Design doc stored late flush[subsystem]=
	Design late rework outgoing rate[subsystem]
	~	Page/Day
	~		|

Design doc stored flush[subsystem]=
	IF THEN ELSE(Rework design or not=1:AND:Portion of design to rework[subsystem]<(Design doc stored size\
		[subsystem]/TIME STEP),Portion of design to rework[subsystem],0)
	~	Page/Day
	~		|

Portion of design to rework[subsystem]=
	Actual design size to develop[subsystem]*Portion of code reworked to actual subsystem code size per subsystem\
		[subsystem]/TIME STEP
	~	Page/Day
	~		|
Actual design defects detected during design verification[origin,factor,subsystem]= INTEG\
		 (
	Design fault detection due to verification[origin,factor,subsystem],
		0)
	~	Defect
	~		|

Design late rework outgoing rate[subsystem]=
	IF THEN ELSE(Portion of design to rework[subsystem]<(Design doc stored size[subsystem\
		]/TIME STEP),MIN(Design to be reworked later[subsystem]/TIME STEP,(Design doc stored size\
		[subsystem]/TIME STEP)-Portion of design to rework[subsystem]),0)
	~	Page/Day
	~		|

Design late rework incoming rate[subsystem]=
	IF THEN ELSE((Design doc stored size[subsystem]/TIME STEP)<Portion of design to rework\
		[subsystem],Portion of design to rework[subsystem],0)
	~	Page/Day
	~		|

Design doc verified flush[subsystem]=
	IF THEN ELSE(Design doc dev status[subsystem]=1,Design doc verified[subsystem]/TIME STEP\
		,0)
	~	Page/Day
	~		|

Code fault detection per subsystem[origin,factor,subsystem]=
	CUSTOMSUMTHREED(Code fault detection[requ,RLB,MOD1],Subsystem's first module number[\
		SUB1],Number of modules per subsystem[SUB1])
	~	Defect/Day
	~		|
Total code doc dev status per subsystem[subsystem]=
        CUSTOMSUMONED(Code doc dev status[MOD1],Subsystem's first module number[subsystem],Number of modules per subsystem\
                [subsystem])/(2*Number of modules per subsystem[subsystem])
        ~       Dmnl
        ~       This variable specifies the status of code document development/rework for \
                every subsystem on a 0 to 1 status.
        |

Sum design to code faults propagated per subsystem[subsystem]=
        CUSTOMSUMONED(Sum design to code faults propagated per module[MOD1],Subsystem's first module number\
                [subsystem],Number of modules per subsystem[subsystem])
        ~       Defect
        ~               |

Sum undetected code faults in IT per quality factor per subsystem[factor,subsystem]=
        CUSTOMSUMTWOD(Sum undetected code faults in IT per quality factor per module[RLB,MOD1\
                ], Subsystem's first module number[SUB1],Number of modules per subsystem[SUB1])
        ~       Defect
        ~               |

Design to code faults propagated per origin[origin,module]=
        SUM(Design to code faults propagated[origin,factor!,module])
        ~       Defect
        ~               |

Sum detected code faults in IT per quality factor per subsystem[factor,subsystem]=
        CUSTOMSUMTWOD(Sum detected code faults in IT per quality factor per module[RLB,MOD1]\
                , Subsystem's first module number[SUB1],Number of modules per subsystem[SUB1])
        ~       Defect
        ~               |

Sum detected code faults in IT per origin per subsystem[origin,subsystem]=
```

CUSTOMSUMTWOD(Sum detected code faults in IT per origin per module[requ,MOD1], Subsystem's first module number\

[SUB1],Number of modules per subsystem[SUB1])

~ Defect

~ |

Sum design to code faults propagated per origin per subsystem[origin,subsystem]=

CUSTOMSUMTWOD(Design to code faults propagated per origin[requ,MOD1],Subsystem's first module number\

[SUB1],Number of modules per subsystem[SUB1])

~ Defect

~ |

Sum undetected code faults in IT per origin per subsystem[origin,subsystem]=

CUSTOMSUMTWOD(Sum undetected code faults in IT per origin per module[requ,MOD1], Subsystem's first module number\

[SUB1],Number of modules per subsystem[SUB1])

~ Defect

~ |

Sum actual code faults detected in IT per quality factor per subsystem[factor,subsystem\

]=

CUSTOMSUMTWOD(Sum actual code faults detected in IT per quality factor per module[RLB\

,MOD1],Subsystem's first module number[SUB1],Number of modules per subsystem[SUB1])

~ Defect

~ |

Sum actual code faults detected in IT per origin per subsystem[origin,subsystem]=

CUSTOMSUMTWOD(Sum actual code faults detected in IT per origin per module[requ,MOD1]\

, Subsystem's first module number[SUB1],Number of modules per subsystem[SUB1])

~ Defect

~ |

Sum code to be tested in IT per subsystem[subsystem]=

```
	CUSTOMSUMONED(Code to be tested in IT[MOD1],Subsystem's first module number[subsystem\
		], Number of modules per subsystem[subsystem])
	~	KLOC
	~		|

Sum detected code faults in IT per subsystem[subsystem]=
	CUSTOMSUMONED(Sum detected code faults in IT per module[MOD1],Subsystem's first module number\
		[subsystem],Number of modules per subsystem[subsystem])
	~	Defect
	~		|

Sum actual code faults detected in IT per subsystem[subsystem]=
	CUSTOMSUMONED(Sum actual code faults detected in IT per module[MOD1],Subsystem's first module number\
		[subsystem],Number of modules per subsystem[subsystem])
	~	Defect
	~		|

Sum tested code in IT per subsystem[subsystem]=
	CUSTOMSUMONED(Tested code in IT[MOD1],Subsystem's first module number[subsystem], Number of modules per subsystem\
		[subsystem])
	~	KLOC
	~		|

Sum IT rate per subsystem[subsystem]=
	CUSTOMSUMONED(IT rate[MOD1],Subsystem's first module number[subsystem],Number of modules per subsystem\
		[subsystem])
	~	KLOC/Day
	~		|

Sum undetected code faults in IT per subsystem[subsystem]=
	CUSTOMSUMONED(Sum undetected code faults in IT per module[MOD1],Subsystem's first module number\
		[subsystem],Number of modules per subsystem[subsystem])
	~	Defect
	~		|

Sum actual code size to develop per subsystem[subsystem]=
	CUSTOMSUMONED(Actual code size to develop per module[MOD1],Subsystem's first module number\
		[subsystem],Number of modules per subsystem[subsystem])
	~	KLOC
	~		|

Subsystem's first module number[subsystem]=
	0,20,50,62,84
	~	Dmnl
	~	This constant specifies the index of the first module in a subsystem on a \
		zero basis indexing for modules. for example if the first module in the \
		second subsystem is the 21st module, then this value for the second \
		subsystem will be 20..
	|

UT end time=
	Phase end time[UT]
	~	Day
	~	This variable specifies the last day that the unit test phase is active.
	|

Total requ spec effort=
	Actual phase effort[RE]
	~	Day*Person
	~	This variable specifies the amount of effort on requirements specification \
		development/verification/rework activities.
	|
Phase end time[phase]=
        SAMPLE IF TRUE(Phase status[phase]=1,Time,0)
        ~       Day
        ~       This variable specifies the last day that a phase is active.
        |

IT duration=
        Phase Duration[IT]
        ~       Day
        ~       This variable specifies the number of days that the integration test phase \
                is active.
        |

ST duration=
        Phase Duration[ST]
        ~       Day
        ~       This variable specifies the number of days that the system test phase is \
                active.
        |

ST end time=
        Phase end time[ST]
        ~       Day
        ~       This variable specifies the last day that the system test phase is active.
        |

Design duration=
        Phase Duration[DE]
        ~       Day
        ~       This variable specifies the number of days that design phase is active.
        |

Project end time=
        VMAX(Phase end time[phase!])
```

~ Day
~ This variable specifies the last day that the project.
|

Design end time=
Phase end time[DE]
~ Day
~ This variable specifies the last day that the design phase is active(some \
work is being done on the design document).
|

Total design effort=
Actual phase effort[DE]
~ Day*Person
~ This variable specifies the amount of effort on design \
development/verification/rework activities.
|

Requ end time=
Phase end time[RE]
~ Day
~ This variable specifies the last day that the requiremens specification \
phase is active(some work is being done on the requirements specification \
document).
|

Requ duration=
Phase Duration[RE]
~ Day
~ This variable specifies the number of days that the requirements \
specification phase is active.
|

Code end time=

```
	Phase end time[CO]
	~	Day
	~	This variable specifies the last day that the code phase is active(some \
		work is being done on the code document).
	|

IT end time=
	Phase end time[IT]
	~	Day
	~	This variable specifies the last day that the integration test phase is \
		active.
	|

Code duration=
	Phase Duration[CO]
	~	Day
	~	This variable specifies the number of days that the code phase is active.
	|

UT duration=
	Phase Duration[UT]
	~	Day
	~	This variable specifies the number of days that the unit test phase is \
		active.
	|

Requ spec ver effectiveness constant=
	0.75
	~	Dmnl
	~	This variable is used to enable changing the value of 'Maximum requ spec \
		ver effectiveness' automatically for all three subscripts (in case of \
		using a 'Sensitivity Analysis'.
	|
Total code effort=
        Actual phase effort[CO]
        ~       Day*Person
        ~       This variable specifies the amount of effort on code \
                development/verification/rework activities.
        |

code fault detection during code verification[origin,factor,module]=
        IF THEN ELSE(Code verification activity[module]>0,Code fault detection[origin,factor\
                ,module],0)
        ~       Defect/Day
        ~               |

Requ defects detected during CI=
        SUM(Sum actual code faults detected during code verification per system[requ,factor!\
                ])/(Average design to code fault multiplier[requ]*Average requ to design fault multiplier\
                [requ])
        ~       Defect
        ~               |

Requ defects detected during DI=
        SUM(Sum actual design defects detected during design verification[requ,factor!])/Average requ to design fault multiplier\
                [requ]
        ~       Defect
        ~               |

Requ defects detected during IT=
        Sum actual code faults detected in IT per origin per system[requ]/(Average design to code fault multiplier\
                [requ]*Average requ to design fault multiplier[requ])
        ~       Defect
        ~               |
Requu defects detected during RI=
        SUM(Actual requ spec faults detected during requ verification[requ,factor!])
        ~       Defect
        ~               |

Requ defects detected during ST=
        Sum actual code faults detected in ST per origin per system[requ]/(Average design to code fault multiplier\
                [requ]*Average requ to design fault multiplier[requ])
        ~       Defect
        ~               |

Actual code faults detected during code verification[origin,factor,module]= INTEG (
        code fault detection during code verification[origin,factor,module],
                0)
        ~       Defect
        ~               |

Design defects detected during CI=
        (SUM(Sum actual code faults detected during code verification per system[design,factor\
                !])+SUM(Sum actual code faults detected during code verification per system[requ,factor\
                !]))/Average design to code fault multiplier[design]
        ~       Defect
        ~               |

Design defects detected during DI=
        SUM(Sum actual design defects detected during design verification[origin!,factor!])
        ~       Defect
        ~               |

Design defects detected during UT=
        (Sum actual code faults detected in UT per origin per system[design]+Sum actual code faults detected in UT per origin per system\
                [requ])/Average design to code fault multiplier[design]
```

```
	~	Defect
	~		|

Code defects detected during code verification=
	SUM(Sum actual code faults detected during code verification per system[origin!,factor\
		!])
	~	Defect
	~		|

Actual requ spec faults detected during requ verification[origin,factor]= INTEG (
	Actual requ spec faults detected during requ verification rate[origin,factor],
		0)
	~	Defect
	~	This level variable specifies the number of requirements specification \
		faults which were detected during the requirements specificaiton \
		verification activity.
	|

Total number of design defects detected=
	Design defects detected during CI+Design defects detected during DI+Design defects detected during IT\
		+Design defects detected during ST+Design defects detected during UT
	~	Defect
	~		|

Total number of detected defects=
	Code defects detected during code verification+Code defects detected during IT+Code defects detected during ST\
		+Code defects detected during UT
	~	Defect
	~		|

Sum actual code faults detected during code verification per system[origin,factor]=
	SUM(Actual code faults detected during code verification[origin,factor,module!])
	~	Defect
	~		|

Sum actual design defects detected during design verification[origin,factor]=
	SUM(Actual design defects detected during design verification[origin,factor,subsystem\
		!])
	~	Defect
	~	This variable sums the number of design defects detected during the design \
		verification activity over  all subsystems.
	|

Total number of detected defects in requ=
	Requ defects detected during CI+Requ defects detected during DI+Requ defects detected during IT\
		+Requ defects detected during RI+Requ defects detected during ST+Requ defects detected during UT
	~	Defect
	~		|

Code defects detected during UT=
	SUM(Sum actual code faults detected in UT per origin per system[origin!])
	~	Defect
	~		|

Code defects detected during ST=
	SUM(Sum actual code faults detected in ST per origin per system[origin!])
	~	Defect
	~		|

Actual requ spec faults detected during requ verification rate[origin,factor]=
	IF THEN ELSE(Requ spec verification activity>0,requ spec fault detection rate[origin\
		,factor],0)
	~	Defect/Day
	~	This rate specifies the number of requirements specification faults which \
		are detected everyday during the requirements specification verification \
		activity.
	|

Design defects detected during ST=
	(Sum actual code faults detected in ST per origin per system[design]+Sum actual code faults detected in ST per origin per system\
		[requ])/Average design to code fault multiplier[design]
	~	Defect
	~		|

Requ defects detected during UT=
	Sum actual code faults detected in UT per origin per system[requ]/(Average design to code fault multiplier\
		[requ]*Average requ to design fault multiplier
	[requ])
	~	Defect
	~		|

Design defects detected during IT=
	(Sum actual code faults detected in IT per origin per system[design]+Sum actual code faults detected in IT per origin per system\
		[requ])/Average design to code fault multiplier[design]
	~	Defect
	~		|

Code defects detected during IT=
	SUM(Sum actual code faults detected in IT per origin per system[origin!])
	~	Defect
	~		|

Project Duration= INTEG (
	Project status,
		0)
	~	Day
	~	This variable specifies the number of days that the project is active.
	|
Total project effort=
        SUM(Actual phase effort[phase!])
        ~       Day*Person
        ~       This variable specifies the total amount of effort spent on all project \
                activities.
        |

Times a module has been verified in code rate[module]=
        IF THEN ELSE(Code verification activity[module]>0,Code verification activity[module]\
                /Actual code size to develop per module[module],0)
        ~       Dmnl/Day
        ~       This rate specifies the rate with which the number of times a module has \
                been verified in code increases.
        |

Times a module has been verified in code[module]= INTEG (
        Times a module has been verified in code rate[module],
                0)
        ~       Dmnl
        ~       This level variable specifies the number of times a module has been \
                verified in the code phase.
        |

Code rework effort per fault[module]=
        IF THEN ELSE(Times a module has been tested in ST[module]>0,Code rework effort for ST\
                ,IF THEN ELSE(Times a module has been tested in IT[module]>0:AND:Times a module has been tested in ST\
                [module]=0,Code rework effort for IT,IF THEN ELSE(Times a module has been tested in UT\
                [module]>0:AND:(Times a module has been tested in IT[module]+Times a module has been tested in ST\
                [module])=0,Code rework effort for UT,IF THEN ELSE(Times a module has been verified in code\
		[module]>0:AND:(Times a module has been tested in UT[module]+Times a module has been tested in IT\
		[module]+Times a module has been tested in ST[module])=0,Code rework effort for code inspection\
		,0))))
	~	Day*Person/Defect
	~		|

Actual phase effort[RE]=
	Actual requ spec effort+Requ ver effort ~~|
Actual phase effort[DE]=
	Actual design effort+Design ver effort ~~|
Actual phase effort[CO]=
	Actual code effort+Code ver effort ~~|
Actual phase effort[UT]=
	Unit test effort ~~|
Actual phase effort[IT]=
	Integration testing effort ~~|
Actual phase effort[ST]=
	System testing effort
	~	Day*Person
	~		|

Code rework effort for IT=
	1.08125
	~	Day*Person/Defect
	~		|

Sum code fault correction per module[module]=
	SUM(Code fault correction[origin!,factor!,module])
	~	Defect/Day
	~		|

Code rework effort for code inspection=
	0.3387
	~	Day*Person/Defect
	~		|

Code rework effort for ST=
	5.6225
	~	Day*Person/Defect
	~		|

Code rework effort for UT=
	0.4325
	~	Day*Person/Defect
	~		|

Sum design fault correction per subsystem[subsystem]=
	SUM(Design fault correction[origin!,factor!,subsystem])
	~	Defect/Day
	~		|

Design rework effort per fault[subsystem]=
	IF THEN ELSE(Design productivity learning switch=1,Initial design rework effort per fault\
		*MAX(1,Design productivity learning status[subsystem]^Productivity design learning amplifier\
		),Initial design rework effort per fault)
	~	Day*Person/Defect
	~		|

Initial design rework effort per fault=
	0.29
	~	Day*Person/Defect
	~		|

Requ spec rework effort per fault=
	IF THEN ELSE(Requ productivity learning switch=0,Initial requ spec rework effort per fault\
		,Initial requ spec rework effort per fault*MAX(1,Requ spec productivity learning status\
		^Requ spec productivity learning amplifier))
	~	(Person*Day)/Defect
	~	This variable specifies the amount of effort that has to be spent to fix a \
		requirements specification defect.
	|

Initial requ spec rework effort per fault=
	0.125
	~	(Person*Day)/Defect
	~	This constant specifies the amount of effort that has to be spent to fix a \
		requirements specification defect initially.
	|

Sum requ spec fault correction rate=
	SUM(Requ spec fault correction rate[origin!,factor!])
	~	Defect/Day
	~	This variable is used to show the sum of requirements specification \
		defects fixed over all origins and quality factors everyday.
	|

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.
	|

Sum design fault detection due to verification[origin,factor]=
	SUM(Design fault detection due to verification[origin,factor,subsystem!])
	~	Defect/Day
	~	This variable sums number of design faults detected during design \
		verification over all subsystems.
	|

Design fault detection due to verification[origin,factor,subsystem]=
	IF THEN ELSE(Design verification activity[subsystem]>0,Design fault detection[origin\
		,factor,subsystem],0)
	~	Defect/Day
	~		|

Phase effort[RE]=
	Requ effort ~~|
Phase effort[DE]=
	Design effort ~~|
Phase effort[CO]=
	Code effort ~~|
Phase effort[UT]=
	Unit test effort ~~|
Phase effort[IT]=
	Integration testing effort ~~|
Phase effort[ST]=
	System testing effort
	~	Day*Person
	~		|

Amount of design reworked per subsystem[subsystem]= INTEG (
	Design to rework[subsystem],
		0)
	~	Page
	~		|

Amount of code reworked per module[module]= INTEG (
	Code to rework[module],
		0)
	~	KLOC
	~		|

Amount of design reworked per system=
	SUM(Amount of design reworked per subsystem[subsystem!])
	~	Page
	~		|

Code randomizing multipliers[module]=
	GET XLS CONSTANTS('Workforce.xls','Sheet2','A1')
	~	Dmnl
	~	This constant specifies a random number between 0.5 and 1.5 generated from \
		a uniform distribution using excel for every module and is used to \
		differentiate size of different modules randomly.
	|
Phase Duration[phase]= INTEG (
        Phase status[phase],
                0)
        ~       Day
        ~       This variable specifies the number of days that a phase is active.
        |

Design to be reworked later[subsystem]= INTEG (
        Design late rework incoming rate[subsystem]-Design late rework outgoing rate[subsystem\
                ],
                0)
        ~       Page
        ~               |

Learning amplifier for code fault injection=
        6
        ~       Dmnl
        ~               |

Learning amplifier for design fault injection=
        2
        ~       Dmnl
        ~               |

Learning amplifier for requ fault injection=
        3
        ~       Dmnl
        ~       This constant is used to adjust the effect of learning on injection of \
                defects in the requirements specification document.
        |

Design productivity learning status[subsystem]=
        IF THEN ELSE(Design learning status[subsystem]<1,1,1+(0.5*Design learning status[subsystem\
		]))
	~	Dmnl
	~		|

Requ spec productivity learning status=
	IF THEN ELSE(Requ spec learning status<1,1,1+(0.6*Requ spec learning status))
	~	Dmnl
	~	This variable is used to adjust the effect of 'Requ spec learning status' \
		on the productivity of the requirements specification development/rework.
	|

Code productivity learning status[module]=
	1+(Code learning status[module]-1)*0
	~	Dmnl
	~		|

Design productivity learning switch=
	0
	~	Dmnl
	~		|

Requ productivity learning switch=
	0
	~	Dmnl
	~	This swirch is used to switch on/off the effect of learning on per defect \
		fixing effort for every pending requirements specification defect.
	|

IT test case data available or not=
	0
	~	Dmnl
	~		|

ST test case data available or not=
        0
        ~       Dmnl
        ~               |

Average number of UT test cases per code size unit=
        10
        ~       Testcase/KLOC
        ~               |

UT Test case data availble or not=
        0
        ~       Dmnl
        ~       This constant specifies if data is available for calibration of parameters \
                related to development/execution of unit test cases. If set to 1 unit test \
                case development is done and test cases are stored in the 'UT test cases' \
                level variable and the unit testing rate is calculated according to unit \
                test case execution data and if set to 0, unit test case development is \
                not done and unit testing rate is calculated according to 'Average UT \
                productivity' variable.
        |

Actual allocation[activity,allocation]=
        GETALLOCATIONX(Developer Capabilities[DEV1,RED],Required skill level per activity[RED
        ],Weighed work[RED],12,Total number of developers in the team)
        ~
        ~       This variable specifies the number and skill level average of developers \
                which are assigned to different activities. The NMBR index of the \
                allocation subscript specifies the number of the assigned developers and \
                the SKLL index specifies the skill level average of the assigned \
                developers.
        |

Phase status[RE]=
	IF THEN ELSE(Total requ spec ver status>0:AND:(Total requ spec status+Total requ spec ver status\
		)/2>0:AND:(Total requ spec status
	+Total requ spec ver status
	)/2<1,1,IF THEN ELSE(Total requ spec ver status=0:AND:Total requ spec status>0:AND:Total requ spec status\
		<1,1,0)) ~~|
Phase status[DE]=
	IF THEN ELSE(Total design doc ver status>0:AND:(Total design doc dev status+Total design doc ver status\
		)/2>0:AND:(Total design doc dev status
+Total design doc ver status
	)/2<1,1,IF THEN ELSE(Total design doc ver status=0:AND:Total design doc dev status>0\
		:AND:Total design doc dev status<1,
1,0)) ~~|
Phase status[CO]=
	IF THEN ELSE(Total code doc ver status>0:AND:(Total code doc dev status+Total code doc ver status\
		)/2>0:AND:(Total code doc dev status
+Total code doc ver status
	)/2<1,1,IF THEN ELSE(Total code doc ver status=0:AND:Total code doc dev status>0:AND:\
		Total code doc dev status<1,1,0)) ~~|
Phase status[UT]=
	IF THEN ELSE(Sum module under TC dev in UT>0,1,IF THEN ELSE(Total unit test status>0\
		:AND:Total unit test status<1,1,0)) ~~|
Phase status[IT]=
	IF THEN ELSE(Number of subsystems under TC dev in IT>0,1,IF THEN ELSE(Total integration test status\
		>0:AND:Total integration test status<1,1,0)) ~~|
Phase status[ST]=
	IF THEN ELSE(System under TC dev in ST>0,1,IF THEN ELSE(Total system test status>0:AND:\
		Total system test status<1,1,0))
	~	Dmnl
	~	This variable specifies the status of a phase within the development \
		project. It is 1 if some work is being done on some document in some \
```

activity of that phase and is 0 if otherwise.

|

Number of documents being processed per activity[RED]=
IF THEN ELSE(Requ spec flag=1,1,0) ~~|
Number of documents being processed per activity[REV]=
IF THEN ELSE(Requ spec ver flag=1,1,0) ~~|
Number of documents being processed per activity[DED]=
Number of subsystems being developed in design ~~|
Number of documents being processed per activity[DEV]=
Number of subsystems being verified in design ~~|
Number of documents being processed per activity[COD]=
Number of modules being developed in code ~~|
Number of documents being processed per activity[COV]=
Number of modules being verified in code ~~|
Number of documents being processed per activity[UTV]=
Number of modules waiting to be unit tested ~~|
Number of documents being processed per activity[ITV]=
Number of subsystems waiting for integration testing ~~|
Number of documents being processed per activity[STV]=
Number of systems waiting to be system tested ~~|
Number of documents being processed per activity[ITTC]=
Number of subsystems waiting for TC dev in IT ~~|
Number of documents being processed per activity[STTC]=
IF THEN ELSE(System waiting for TC dev for ST=1,1,0) ~~|
Number of documents being processed per activity[UTTC]=
Number of modules waiting for TC dev in UT
~ Dmnl
~ This variable specifies the number of documents which are waiting to be \
worked on in every activity. The requirements specification document is \
counted as 1 document. A subsystem's design document is counted as 1 \
document. A module's code document is count as 1 document.

|

```
UT test cases[module]= INTEG (
          UT test case dev rate[module],
                    0)
          ~         Testcase
          ~         This level variable specifies the number of test cases developed for unit \
                    test.
          |

Sum allocated personnel=
          SUM(Actual allocation[activity!,NMBR])
          ~         Dmnl
          ~         This variable specifies the total number of developers that are assigned \
                    to all activites within the development process.
          |

Average requ spec ver effectiveness[origin,factor]=
          Maximum requ spec ver effectiveness[origin,factor]*Requ ver team skill level average
          ~         Dmnl
          ~         This variable specifies the effectiveness of the requirements \
                    specification verification technique in detecting undetected requirements \
                    specification defects.
          |

ST test cases= INTEG (
          ST test case dev rate,
                    0)
          ~         Testcase
          ~                   |

allocation:
          NMBR,SKLL
          ~
          ~                   |
IT test cases[subsystem]= INTEG (
          IT test case dev rate[subsystem],
                    0)
          ~         Testcase
          ~                   |

Times a module has been tested in UT rate[module]=
          IF THEN ELSE(Code returned for rework rate from UT[module]>0,1,0)
          ~         Dmnl/Day
          ~         This rate specifies the value of times a module has been tested in UT that \
                    is added everyday.
          |

Times a module has been tested in UT[module]= INTEG (
          Times a module has been tested in UT rate[module],
                    0)
          ~         Dmnl
          ~         This level variable specifies the number of times a module has been tested \
                    in UT.
          |

Times a module has been tested in IT rate[module]=
          IF THEN ELSE(Code returned for rework from IT[module]>0,1,0)
          ~         Dmnl/Day
          ~                   |

Times a module has been tested in IT[module]= INTEG (
          Times a module has been tested in IT rate[module],
                    0)
          ~         Dmnl
          ~                   |

Times a module has been tested in ST[module]= INTEG (
          Times a module has been tested in ST rate[module],
		0)
	~	Dmnl
	~		|

Times a module has been tested in ST rate[module]=
	IF THEN ELSE(Code returned for rework from ST[module]>0,1,0)
	~	Dmnl/Day
	~		|

Required skill level per activity[activity]=
	0,0.8,0,0,0,0,0,0,0,0,0,0,0
	~	Dmnl
	~	This constant specifies the skill level which is required for developers \
		in order to be assigned to an activity.
	|

Code weigh for dev=
	1
	~	Dmnl
	~	This variable specifies the weight of a module's code document while its \
		waiting to be developed/reworked.
	|

Project status=
	IF THEN ELSE(SUM(Phase status[phase!])>0,1,0)
	~	Dmnl
	~	This variable specifies the status of the project. It is 1 if some phase \
		is active and 0 if otherwise.
	|

Code weigh for ver=
	1
	~	Dmnl
	~	This variable specifies the weight of a module's code document while its \
		waiting to be verified.
	|

Document type weighs[RED]=
	Requ spec weigh for dev ~~|
Document type weighs[REV]=
	Requ spec weigh for ver ~~|
Document type weighs[DED]=
	Design weigh for dev ~~|
Document type weighs[DEV]=
	Design weigh for ver ~~|
Document type weighs[COD]=
	Code weigh for dev ~~|
Document type weighs[COV]=
	Code weigh for ver ~~|
Document type weighs[UTV]=
	Code weigh for ut ~~|
Document type weighs[ITV]=
	Design weigh for it ~~|
Document type weighs[STV]=
	Requ spec weigh for st ~~|
Document type weighs[ITTC]=
	Design weigh for it ~~|
Document type weighs[STTC]=
	Requ spec weigh for st ~~|
Document type weighs[UTTC]=
	Code weigh for ut
	~	Dmnl
	~	Since the allocation algorithm uses the number of documents which are \
		waiting to be worked on, this variable is used to weigh different document \
		types so that for example the requirements specification is considered \
		equal to a couple of modules' code document.
	|
Requ spec weigh for st=
        1
        ~       Dmnl
        ~       This variable specifies the weight of a module's code document while its \
                waiting to be system tested.
        |

Design weigh for dev=
        1
        ~       Dmnl
        ~       This variable specifies the weight of a subsystem's design document while \
                its waiting to be developed/reworked.
        |

Design weigh for it=
        1
        ~       Dmnl
        ~       This variable specifies the weight of a module's code document while its \
                waiting to be integration tested.
        |

Requ spec weigh for dev=
        1
        ~       Dmnl
        ~       This variable specifies the weight of the requirements specification \
                document while its waiting to be developed/reworked.
        |

Requ spec weigh for ver=
        1
        ~       Dmnl
        ~       This variable specifies the weight of the requirements specification \
                document while its waiting to be verified.
        |
Design weigh for ver=
	1
	~	Dmnl
	~	This variable specifies the weight of a subsystem's design document while \
		its waiting to be verified.
	|

Code weigh for ut=
	1
	~	Dmnl
	~	This variable specifies the weight of a module's code document while its \
		waiting to be unit tested.
	|

Weighed work[activity]=
	Document type weighs[activity]*Number of documents being processed per activity[activity\
		]
	~	Dmnl
	~	This variable specifies the weighed number of documents that are waiting \
		to be processed in every activity.
	|

activity:
	RED,REV,DED,DEV,COD,COV,UTTC,UTV,ITTC,ITV,STTC,STV
	~
	~		|

co activities:
	COD,COV
	~
	~		|

it activities:
	ITV,ITTC
	~	
	~		|

st activities:
	STTC,STV
	~	
	~		|

ut activities:
	UTTC,UTV
	~	
	~		|

de activities:
	DED,DEV
	~	
	~		|

re activities:
	RED,REV
	~	
	~		|

developer:
	(DEV1-DEV40)
	~	
	~		|

Developer Capabilities[developer,activity]=
	GET XLS CONSTANTS('Workforce.xls','Sheet1','A1')
	~	Dmnl
	~	This constant is in the form of a matrix with one row for every developer \
		and one column for every activity. The cell at the ith row and jth column \
		specifies the skill level of developer i in carrying out the activity j on \
		a 0 to 1 basis.
	|

Normalized requ spec quality measures per system per quality factor[factor]=
	IF THEN ELSE(Sum requ spec faults undetected per quality factor[factor]>0,Sum actual requ spec faults corrected per quality factor\
		[factor]/(Sum actual requ spec faults detected per quality factor[factor]+Sum requ spec faults undetected per quality factor
	[factor]),0)
	~	Dmnl
	~		|

Normalized code quality measures per system per quality factor[factor]=
	IF THEN ELSE(Sum code faults undetected in coding per quality factor per system[factor\
		]>0,Sum actual code faults corrected per quality factor per system
	[factor]/(Sum actual code faults detected per quality factor per system[factor]+Sum code faults undetected in coding per quality factor per system
	[factor]),0)
	~	Dmnl
	~		|

Normalized design measure per system=
	IF THEN ELSE((Sum actual design faults detected per system+Sum design faults undetected per system\
		)>0,Sum actual design faults corrected per system
	/(Sum actual design faults detected per system+Sum design faults undetected per system\
		),0)
	~	Dmnl
	~		|

Normalized code quality measure per system=
	IF THEN ELSE((Sum actual code faults detected per system+Sum code faults undetected per system\
		)>0,Sum actual code faults corrected per system
	/(Sum actual code faults detected per system+Sum code faults undetected per system),\
		0)
	~	Dmnl
	~		|

Normalized design quality measures for system per quality factor[factor]=
	IF THEN ELSE(Sum design faults undetected per quality factor per system[factor]>0,Sum actual design faults corrected per quality factor per system\
		[factor]/(Sum actual design faults detected per quality factor per system[factor]+Sum design faults undetected per quality factor per system
	[factor]),0)
	~	Dmnl
	~		|

Verify design or not=
	1
	~	Dmnl
	~		|

Verify code or not=
	1
	~	Dmnl
	~		|

System test or not=
	1
	~	Dmnl
	~	This variable is used as a switch to turn on/off the system test activity. \
		While set to 1 system test is done and while set to 0 its not.
	|

Integration test or not=
	1
	~	Dmnl
	~	This variable is used as a switch to turn on/off the integration test \
		activity. While set to 1 integration test is done and while set to 0 its \
		not.
	|

Unit test or not=
	1
	~	Dmnl
	~	This variable is used as a switch to turn on/off the unit test activity. \
		While set to 1 unit test is done and while set to 0 its not.
	|

Design doc quality[subsystem]=
	IF THEN ELSE(Design doc verified[subsystem]>0,Sum design faults pending per subsystem\
		[subsystem]/Design doc verified[subsystem], 0)
	~	Defect/Page
	~		|

Design doc verified[subsystem]= INTEG (
	Design verification activity[subsystem]-Design doc verified flush[subsystem],
		0)
	~	Page
	~		|

Sum design to code faults propagated per module[module]=
	SUM(Design to code faults propagated[origin!,factor!,module])
	~	Defect
	~		|

Code dev flag[module]=
	IF THEN ELSE(Code doc dev status[module]=1,1,0)
	~	Dmnl
	~		|

Number of subsystems being verified in design=
	SUM(Design ver flag[subsystem!])
	~	Dmnl
	~	This variable specifies the number of subsystems being verified at any \
		point in time.
	|

Code ver flag[module]=
	IF THEN ELSE(Code doc ver status[module]=1,1,0)
	~	Dmnl
	~		|

Design dev flag[subsystem]=
	IF THEN ELSE(Design doc dev status[subsystem]=1,1,0)
	~	Dmnl
	~		|

Design ver flag[subsystem]=
	IF THEN ELSE(Design doc ver status[subsystem]=1,1,0)
	~	Dmnl
	~		|

Total number of developers in the team=
	40
	~	Person
	~	This constant specifies the total number of developers that are available \
		for the project.
	|

Requ spec flag=
	IF THEN ELSE(Requ spec status=1,1,0)
	~	Dmnl
	~	This flag is used to specify when the requirements specification \
		development/rework activity is active.
	|
Requ spec ver flag=
        IF THEN ELSE(Requ spec ver status=1,1,0)
        ~       Dmnl
        ~       This flag is used to specify when the requirements specification \
                verification is active.
        |

Number of subsystems being developed in design=
        SUM(Design dev flag[subsystem!])
        ~       Dmnl
        ~       This variable specifies the number of subsystems being developed/reworked \
                at any point in time.
        |

Number of modules being developed in code=
        SUM(Code dev flag[module!])
        ~       Dmnl
        ~               |

Number of modules being verified in code=
        SUM(Code ver flag[module!])
        ~       Dmnl
        ~               |

"V&V status"[RE]=
        IF THEN ELSE(Total requ spec ver status>0:AND:Total requ spec ver status<1,1,0) ~~|
"V&V status"[DE]=
        IF THEN ELSE(Total design doc ver status>0:AND:Total design doc ver status<1,1,0) ~~|
"V&V status"[CO]=
        IF THEN ELSE(Total code doc ver status>0:AND:Total code doc ver status<1,1,0) ~~|
"V&V status"[UT]=
        IF THEN ELSE(Total unit test status>0:AND:Total unit test status<1,1,0) ~~|
"V&V status"[IT]=
	IF THEN ELSE(Total integration test status>0:AND:Total integration test status<1,1,0\
		) ~~|
"V&V status"[ST]=
	IF THEN ELSE(Total system test status>0:AND:Total system test status<1,1,0)
	~	Dmnl
	~	This variable specifies the status of the verification and validation \
		activites within the project. It is 1 if some V&V activity is active and 0 \
		if otherwise.
	|

Total code doc ver status per subsystem[subsystem]=
	CUSTOMSUMONED(Code doc ver status[MOD1],Subsystem's first module number[subsystem],Number of modules per subsystem\
		[subsystem])/(3*Number of modules per subsystem[subsystem])
	~	Dmnl
	~	This variable specifies the status of code document verification for every \
		subsystem on a 0 to 1 status.
	|

Dev status[RE]=
	IF THEN ELSE(Total requ spec status>0:AND:Total requ spec status<1,1,0) ~~|
Dev status[DE]=
	IF THEN ELSE(Total design doc dev status>0:AND:Total design doc dev status<1,1,0) ~~|
Dev status[CO]=
	IF THEN ELSE(Total code doc dev status>0:AND:Total code doc dev status<1,1,0)
	~	Dmnl
	~	This variable specifies the status of the development/rework activities \
		within the project. It is 1 if some development/rework activity is active \
		and 0 if otherwise.
	|

phase:
	RE,DE,CO,UT,IT,ST -> (activity:re activities,de activities,co activities,ut activities\
		,it activities,st activities)
	~	
	~		|

Total code doc dev status=
	SUM(Code doc dev status[module!])/(2*SUM(Number of modules per subsystem[subsystem!]\
		))
	~	Dmnl
	~		|

Total design doc dev status=
	SUM(Design doc dev status[subsystem!])/(2*Number of subsystems per product)
	~	Dmnl
	~	This variable specifies the overall status of the design \
		development/rework activity on a 0 to 1 basis.
	|

Total requ spec ver status=
	Requ spec ver status/3
	~	Dmnl
	~	This variable is used to specify the status of the requirements \
		specification activity on a 0 to 1 basis.
	|

Total requ spec status=
	Requ spec status/2
	~	Dmnl
	~	This variable shows the status of the requirements specification \
		development/rework activity on a 0 to 1 basis.
	|

Total system test status=
	SUM(System test status[module!])/(3*SUM(Number of modules per subsystem[subsystem!])\
		)
	~	Dmnl
	~		|

Rework design or not=
	1
	~	Dmnl
	~		|

Total design doc ver status=
	SUM(Design doc ver status[subsystem!])/(3*Number of subsystems per product)
	~	Dmnl
	~	This variable specifies the overall status of the design verification \
		activity on a 0 to 1 basis.
	|

Total code doc ver status=
	SUM(Code doc ver status[module!])/(3*SUM(Number of modules per subsystem[subsystem!]\
		))
	~	Dmnl
	~		|

Sum requ spec faults pending per origin[origin]=
	SUM(Requ spec faults pending[origin,factor!])
	~	Defect
	~		|

Sum requ spec faults pending per quality factor[factor]=
	SUM(Requ spec faults pending[origin!,factor])
	~	Defect
	~		|

Sum actual requ spec faults corrected per system=
	SUM(Sum actual requ spec faults corrected per origin[origin!])
	~	Defect
	~		|
Sum requ spec faults generated per system=
	SUM(Sum requ spec faults generated per origin[origin!])
	~	Defect
	~		|

Sum requ spec faults undetected per system=
	SUM(Sum requ spec faults undetected per origin[origin!])
	~	Defect
	~	This variable the overall number of defects that remain undetected in the \
		requirements specification document.
	|

Sum actual requ spec faults detected per system=
	SUM(Sum actual requ spec faults detected per origin[origin!])
	~	Defect
	~	This variable specifies the total number of defects which were detected in \
		the requirements specification document.
	|

Sum requ spec faults pending per system=
	SUM(Sum requ spec faults pending per origin[origin!])
	~	Defect
	~		|

Sum requ spec faults generated per origin[origin]=
	SUM(Requ spec faults generated[origin,factor!])
	~	Defect
	~		|

Sum actual design faults detected per quality factor per system[factor]=
	SUM(Sum actual design faults detected per quality factor per subsystem[factor,subsystem\
		!])
	~	Defect
	~		|

Sum requ spec faults generated per quality factor[factor]=
	SUM(Requ spec faults generated[origin!,factor])
	~	Defect
	~		|

Sum requ spec faults undetected per origin[origin]=
	SUM(Requ spec faults undetected[origin,factor!])
	~	Defect
	~		|

Sum requ spec faults undetected per quality factor[factor]=
	SUM(Requ spec faults undetected[origin!,factor])
	~	Defect
	~		|

Sum actual requ spec faults detected per quality factor[factor]=
	SUM(Actual requ spec faults detected[origin!,factor])
	~	Defect
	~		|

Sum design faults pending per quality factor per system[factor]=
	SUM(Sum design faults pending per quality factor per subsystem[factor,subsystem!])
	~	Defect
	~		|

Sum design faults pending per origin per system[origin]=
	SUM(Sum design faults pending per origin per subsystem[origin,subsystem!])
	~	Defect
	~		|

Sum actual requ spec faults corrected per origin[origin]=
	SUM(Actual requ spec faults corrected[origin,factor!])
```

```
        ~       Defect
        ~               |

Sum actual requ spec faults detected per origin[origin]=
        SUM(Actual requ spec faults detected[origin,factor!])
        ~       Defect
        ~               |

Sum actual design faults detected per origin per system[origin]=
        SUM(Sum actual design faults detected per origin per subsystem[origin,subsystem!])
        ~       Defect
        ~               |

Sum actual requ spec faults corrected per quality factor[factor]=
        SUM(Actual requ spec faults corrected[origin!,factor])
        ~       Defect
        ~               |

Sum design faults undetected per origin per system[origin]=
        SUM(Sum design faults undetected per origin per subsystem[origin,subsystem!])
        ~       Defect
        ~               |

Sum design faults undetected per quality factor per subsystem[factor,subsystem]=
        SUM(Design faults undetected[origin!,factor,subsystem])
        ~       Defect
        ~               |

Sum design faults undetected per quality factor per system[factor]=
        SUM(Sum design faults undetected per quality factor per subsystem[factor,subsystem!]\
                )
        ~       Defect
        ~               |
Sum code faults generated per quality factor per system[factor]=
        SUM(Sum code faults generated per quality factor per module[factor,module!])
        ~       Defect
        ~               |

Sum actual design faults corrected per origin per system[origin]=
        SUM(Sum actual design faults corrected per origin per subsystem[origin,subsystem!])
        ~       Defect
        ~               |

Sum actual design faults corrected per quality factor per subsystem[factor,subsystem]\
                =
        SUM(Actual design faults corrected[origin!,factor,subsystem])
        ~       Defect
        ~               |

Sum actual design faults corrected per quality factor per system[factor]=
        SUM(Sum actual design faults corrected per quality factor per subsystem[factor,subsystem\
                !])
        ~       Defect
        ~               |

Sum code faults pending per quality factor per system[factor]=
        SUM(Sum code faults pending per quality factor per module[factor,module!])
        ~       Defect
        ~               |

Sum actual code faults detected per quality factor per system[factor]=
        SUM(Sum actual code faults detected per quality factor per module[factor,module!])
        ~       Defect
        ~               |

Sum code faults pernding per origin per system[origin]=
        SUM(Sum code faults pending per origin per module[origin,module!])
```

~ Defect

~ |

Sum actual design faults detected per quality factor per subsystem[factor,subsystem]=

SUM(Actual design faults detected[origin!,factor,subsystem])

~ Defect

~ |

Sum code faults undetected in coding per origin per system[origin]=

SUM(Sum code faults undetected in coding per origin per module[origin,module!])

~ Defect

~ |

Sum actual code faults corrected per origin per system[origin]=

SUM(Sum actual code faults corrected per origin per module[origin,module!])

~ Defect

~ |

Sum actual code faults corrected per quality factor per module[factor,module]=

SUM(Actual code faults corrected[origin!,factor,module])

~ Defect

~ |

Sum design faults generated per origin per system[origin]=

SUM(Sum design faults generated per origin per subsystem[origin,subsystem!])

~ Defect

~ |

Sum design faults generated per quality factor per subsystem[factor,subsystem]=

SUM(Design faults generated[origin!,factor,subsystem])

~ Defect

~ |

Sum design faults generated per quality factor per system[factor]=

```
	SUM(Sum design faults generated per quality factor per subsystem[factor,subsystem!])
	~	Defect
	~		|

Sum code faults generated per quality factor per module[factor,module]=
	SUM(Code faults generated[origin!,factor,module])
	~	Defect
	~		|

Sum actual code faults detected per origin per system[origin]=
	SUM(Sum actual code faults detected per origin per module[origin,module!])
	~	Defect
	~		|

Sum actual code faults detected per quality factor per module[factor,module]=
	SUM(Actual code faults detected[origin!,factor,module])
	~	Defect
	~		|

Sum design faults pending per quality factor per subsystem[factor,subsystem]=
	SUM(Design faults pending[origin!,factor,subsystem])
	~	Defect
	~		|

Sum code faults generated per origin per system[origin]=
	SUM(Sum code faults generated per origin per module[origin,module!])
	~	Defect
	~		|

Sum code faults undetected in coding per quality factor per module[factor,module]=
	SUM(Code faults undetected in coding[origin!,factor,module])
	~	Defect
	~		|
Sum code faults undetected in coding per quality factor per system[factor]=
        SUM(Sum code faults undetected in coding per quality factor per module[factor,module\
                !])
        ~       Defect
        ~               |

Sum actual code faults corrected per quality factor per system[factor]=
        SUM(Sum actual code faults corrected per quality factor per module[factor,module!])
        ~       Defect
        ~               |

Sum code faults pending per quality factor per module[factor,module]=
        SUM(Code faults pending[origin!,factor,module])
        ~       Defect
        ~               |

Sum actual code faults detected in ST per origin per module[origin,module]=
        SUM(Actual code faults detected in ST[origin,factor!,module])
        ~       Defect
        ~               |

Sum actual code faults detected in IT per module[module]=
        SUM(Actual code faults detected in IT[origin!,factor!,module])
        ~       Defect
        ~               |

Sum actual code faults detected in IT per origin per module[origin,module]=
        SUM(Actual code faults detected in IT[origin,factor!,module])
        ~       Defect
        ~               |

Sum actual code faults detected in IT per origin per system[origin]=
        SUM(Sum actual code faults detected in IT per origin per subsystem[origin,subsystem!\
                ])
```

```
	~	Defect
	~		|

Sum actual code faults detected in IT per quality factor per module[factor,module]=
	SUM(Actual code faults detected in IT[origin!,factor,module])
	~	Defect
	~		|

Sum actual code faults detected in IT per quality factor per system[factor]=
	SUM(Sum actual code faults detected in IT per quality factor per subsystem[factor,subsystem\
		!])
	~	Defect
	~		|

Sum actual code faults detected in IT per system=
	SUM(Sum actual code faults detected in IT per subsystem[subsystem!])
	~	Defect
	~		|

Sum undetected code faults in UT per origin per system[origin]=
	SUM(Sum undetected code faults in UT per origin per module[origin,module!])
	~	Defect
	~		|

Sum actual code faults detected in ST per origin per system[origin]=
	SUM(Sum actual code faults detected in ST per origin per module[origin,module!])
	~	Defect
	~		|

Sum actual code faults detected in ST per quality factor per module[factor,module]=
	SUM(Actual code faults detected in ST[origin!,factor,module])
	~	Defect
	~		|
Sum actual code faults detected in ST per quality factor per system[factor]=
        SUM(Sum actual code faults detected in ST per quality factor per module[factor,module\
                !])
        ~       Defect
        ~               |

Sum actual code faults detected in ST per system=
        SUM(Sum actual code faults detected in ST per module[module!])
        ~       Defect
        ~               |

Sum undetected code faults in ST per origin per system[origin]=
        SUM(Sum undetected code faults in ST per origin per module[origin,module!])
        ~       Defect
        ~               |

Sum undetected code faults in ST per quality factor pe rmodule[factor,module]=
        SUM(Undetected code faults in ST[origin!,factor,module])
        ~       Defect
        ~               |

Sum actual code faults detected in UT per origin per system[origin]=
        SUM(Sum actual code faults detected in UT per origin per module[origin,module!])
        ~       Defect
        ~               |

Sum undetected code faults in ST per system=
        SUM(Sum undetected code faults in ST per module[module!])
        ~       Defect
        ~               |

Sum actual code faults detected in UT per quality factor per system[factor]=
        SUM(Sum actual code faults detected in UT per quality factor per module[factor,module\
                !])
```

```
	~	Defect
	~		|

Sum detected code faults in IT per system=
	SUM(Sum detected code faults in IT per subsystem[subsystem!])
	~	Defect
	~		|

Sum undetected code faults in IT per quality factor per module[factor,module]=
	SUM(Undetected code faults in IT[origin!,factor,module])
	~	Defect
	~		|

Sum detected code faults in ST per module[module]=
	SUM(Detected code faults in ST[origin!,factor!,module])
	~	Defect
	~		|

Sum undetected code faults in UT per quality factor per system[factor]=
	SUM(Sum undetected code faults in UT per quality factor per module[factor,module!])
	~	Defect
	~		|

Sum detected code faults in IT per origin per module[origin,module]=
	SUM(Detected code faults in IT[origin,factor!,module])
	~	Defect
	~		|

Sum detected code faults in IT per origin per system[origin]=
	SUM(Sum detected code faults in IT per origin per subsystem[origin,subsystem!])
	~	Defect
	~		|

Sum detected code faults in IT per quality factor per module[factor,module]=
	SUM(Detected code faults in IT[origin!,factor,module])
	~	Defect
	~		|

Sum detected code faults in IT per quality factor per system[factor]=
	SUM(Sum detected code faults in IT per quality factor per subsystem[factor,subsystem\
		!])
	~	Defect
	~		|

Sum tested code in ST per system=
	SUM(Tested code in ST[module!])
	~	KLOC
	~		|

Sum undetected code faults in IT per origin per system[origin]=
	SUM(Sum undetected code faults in IT per origin per subsystem[origin,subsystem!])
	~	Defect
	~		|

Sum detected code faults in ST per quality factor per system[factor]=
	SUM(Sum detected code faults in ST per quality factor per module[factor,module!])
	~	Defect
	~		|

Sum detected code faults in ST per system=
	SUM(Sum detected code faults in ST per module[module!])
	~	Defect
	~		|

Sum detected code faults in ST per module per origin[origin,module]=
	SUM(Detected code faults in ST[origin,factor!,module])
	~	Defect
	~		|
Sum detected code faults in ST per origin per system[origin]=
        SUM(Sum detected code faults in ST per module per origin[origin,module!])
        ~       Defect
        ~               |

Sum detected code faults in ST per quality factor per module[factor,module]=
        SUM(Detected code faults in ST[origin!,factor,module])
        ~       Defect
        ~               |

Sum code to be tested in ST per system=
        SUM(Code to be tested in ST[module!])
        ~       KLOC
        ~               |

Sum undetected code faults in ST per origin per module[origin,module]=
        SUM(Undetected code faults in ST[origin,factor!,module])
        ~       Defect
        ~               |

Sum detected code faults in UT per qualit factor per system[factor]=
        SUM(Sum detected code faults in UT per quality factor per module[factor,module!])
        ~       Defect
        ~               |

Sum detected code faults in UT per origin per system[origin]=
        SUM(Sum detected code faults in UT per origin per module[origin,module!])
        ~       Defect
        ~               |

Sum undetected code faults in ST per quality factor per system[factor]=
        SUM(Sum undetected code faults in ST per quality factor pe rmodule[factor,module!])
        ~       Defect
```

```
	~		|

Sum undetected code faults in IT per system=
	SUM(Sum undetected code faults in IT per subsystem[subsystem!])
	~	Defect
	~		|

Sum undetected code faults in IT per quality factor per system[factor]=
	SUM(Sum undetected code faults in IT per quality factor per subsystem[factor,subsystem\
		!])
	~	Defect
	~		|

Sum undetected code faults in IT per origin per module[origin,module]=
	SUM(Undetected code faults in IT[origin,factor!,module])
	~	Defect
	~		|

Sum detected code faults in UT per quality factor per module[factor,module]=
	SUM(Detected code faults in UT[origin!,factor,module])
	~	Defect
	~		|

Sum undetected code faults in UT per system=
	SUM(Sum undetected code faults in UT per module[module!])
	~	Defect
	~		|

Sum detected code faults in UT per system=
	SUM(Sum detected code faults in UT per module[module!])
	~	Defect
	~		|

Sum detected code faults in UT per module[module]=
	SUM(Detected code faults in UT[origin!,factor!,module])
	~	Defect
	~		|

Sum detected code faults in UT per origin per module[origin,module]=
	SUM(Detected code faults in UT[origin,factor!,module])
	~	Defect
	~		|

Sum undetected code faults in UT per module[module]=
	SUM(Undetected code faults in UT[origin!,factor!,module])
	~	Defect
	~		|

Sum undetected code faults in UT per quality factor per module[factor,module]=
	SUM(Undetected code faults in UT[origin!,factor,module])
	~	Defect
	~		|

Sum undetected code faults in UT per origin per module[origin,module]=
	SUM(Undetected code faults in UT[origin,factor!,module])
	~	Defect
	~		|

Sum actual code faults detected in UT per quality factor per module[factor,module]=
	SUM(Actual code faults detected in UT[origin!,factor,module])
	~	Defect
	~		|

factor:
	RLB,USB,FUN
	~
	~		|
Sum actual code faults detected in ST per module[module]=
        SUM(Actual code faults detected in ST[origin!,factor!,module])
        ~       Defect
        ~               |

Sum undetected code faults in IT per module[module]=
        SUM(Undetected code faults in IT[origin!,factor!,module])
        ~       Defect
        ~               |

Sum detected code faults in IT per module[module]=
        SUM(Detected code faults in IT[origin!,factor!,module])
        ~       Defect
        ~               |

Sum undetected code faults in ST per module[module]=
        SUM(Undetected code faults in ST[origin!,factor!,module])
        ~       Defect
        ~               |

Sum tested code in IT per system=
        SUM(Tested code in IT[module!])
        ~       KLOC
        ~               |

Sum code to be tested in IT per system=
        SUM(Code to be tested in IT[module!])
        ~       KLOC
        ~               |

Sum actual code size to develop per system=
        SUM(Actual code size to develop per module[module!])
        ~       KLOC
        ~               |
Sum tested code in UT per system=
	SUM(Tested code in UT[module!])
	~	KLOC
	~		|

Sum code to be tested in UT per system=
	SUM(Code to be tested in UT[module!])
	~	KLOC
	~		|

Sum actual code faults detected in UT per module[module]=
	SUM(Actual code faults detected in UT[origin!,factor!,module])
	~	Defect
	~		|

Sum actual code faults detected in UT per origin per module[origin,module]=
	SUM(Actual code faults detected in UT[origin,factor!,module])
	~	Defect
	~		|

Sum actual code faults detected in UT per system=
	SUM(Sum actual code faults detected in UT per module[module!])
	~	Defect
	~		|

Sum code faults generated per system=
	SUM(Sum code faults generated per module[module!])
	~	Defect
	~		|

Sum code faults generated per origin per module[origin,module]=
	SUM(Code faults generated[origin,factor!,module])
	~	Defect
	~		|

Sum design faults generated per origin per subsystem[origin,subsystem]=
	SUM(Design faults generated[origin,factor!,subsystem])
	~	Defect
	~		|

Sum actual code faults corrected per system=
	SUM(Sum actual code faults corrected per module[module!])
	~	Defect
	~		|

Sum actual code faults corrected per origin per module[origin,module]=
	SUM(Actual code faults corrected[origin,factor!,module])
	~	Defect
	~		|

Sum actual design faults corrected per origin per subsystem[origin,subsystem]=
	SUM(Actual design faults corrected[origin,factor!,subsystem])
	~	Defect
	~		|

Sum actual code faults detected per system=
	SUM(Sum actual code faults detected per module[module!])
	~	Defect
	~	This variable specifies the total number of defects which were detected in \
		the code document.
	|

Sum actual code faults detected per origin per module[origin,module]=
	SUM(Actual code faults detected[origin,factor!,module])
	~	Defect
	~		|
Sum actual design faults detected per origin per subsystem[origin,subsystem]=
        SUM(Actual design faults detected[origin,factor!,subsystem])
        ~       Defect
        ~               |

Sum code faults pending per origin per module[origin,module]=
        SUM(Code faults pending[origin,factor!,module])
        ~       Defect
        ~               |

Sum design faults pending per origin per subsystem[origin,subsystem]=
        SUM(Design faults pending[origin,factor!,subsystem])
        ~       Defect
        ~               |

Sum design faults undetected per origin per subsystem[origin,subsystem]=
        SUM(Design faults undetected[origin,factor!,subsystem])
        ~       Defect
        ~               |

Sum code faults undetected in coding per origin per module[origin,module]=
        SUM(Code faults undetected in coding[origin,factor!,module])
        ~       Defect
        ~               |

Sum design doc stored size per system=
        SUM(Design doc stored size[subsystem!])
        ~       Page
        ~               |

Sum code to do size per system=
        SUM(Code to do size[module!])
        ~       KLOC
        ~               |
Sum design doc ready size per system=
	SUM(Design doc ready size[subsystem!])
	~	Page
	~		|

Sum design doc size per system=
	SUM(Design doc size[subsystem!])
	~	Page
	~		|

Sum actual design faults detected per system=
	SUM(Sum actual design faults detected per subsystem[subsystem!])
	~	Defect
	~	This variable specifies the total number of defects which were detected in \
		the design document.
	|

Sum code doc ready size per system=
	SUM(Code doc ready size[module!])
	~	KLOC
	~		|

Sum code doc stored size per system=
	SUM(Code doc stored size[module!])
	~	KLOC
	~		|

Sum actual code faults detected per module[module]=
	SUM(Actual code faults detected[origin!,factor!,module])
	~	Defect
	~		|

Sum code faults undetected per system=
	SUM(Sum code faults undetected per module[module!])
	~	Defect
	~	This variable the overall number of defects that remain undetected in the \
		code document.
	|

Sum actual code faults corrected per module[module]=
	SUM(Actual code faults corrected[origin!,factor!,module])
	~	Defect
	~		|

Sum code faults pending per system=
	SUM(Sum code faults pending per module[module!])
	~	Defect
	~		|

Sum design to do size per system=
	SUM(Design to do size[subsystem!])
	~	Page
	~		|

Sum design faults pending per system=
	SUM(Sum design faults pending per subsystem[subsystem!])
	~	Defect
	~		|

Sum code faults generated per module[module]=
	SUM(Code faults generated[origin!,factor!,module])
	~	Defect
	~		|

Sum design faults generated per system=
	SUM(Sum design faults generated per subsystem[subsystem!])
	~	Defect
	~		|

Sum code doc size per system=
	SUM(Code doc size[module!])
	~	KLOC
	~		|

Sum actual design faults corrected per system=
	SUM(Sum actual design faults corrected per subsystem[subsystem!])
	~	Defect
	~		|

Sum design faults undetected per system=
	SUM(Sum design faults undetected per subsystem[subsystem!])
	~	Defect
	~	This variable the overall number of defects that remain undetected in the \
		design document.
	|

module:
	(MOD1-MOD100)
	~
	~		|

mod sub1: (MOD1-MOD20)
	~
	~		|

mod sub2: (MOD21-MOD50)
	~
	~		|

mod sub3: (MOD51-MOD62)
	~
	~		|

mod sub4: (MOD63-MOD84)
	~
	~		|

mod sub5: (MOD85-MOD100)
	~
	~		|

subsystem:
	(SUB1-SUB5) -> (module:mod sub1, mod sub2, mod sub3, mod sub4, mod sub5)
	~
	~		|

origin:
	requ, design, code
	~
	~		|

Sum code faults undetected per module[module]=
	SUM(Code faults undetected in coding[origin!,factor!,module])
	~	Defect
	~		|

Sum actual design faults corrected per subsystem[subsystem]=
	SUM(Actual design faults corrected[origin!,factor!,subsystem])
	~	Defect
	~		|

Sum actual design faults detected per subsystem[subsystem]=
	SUM(Actual design faults detected[origin!,factor!,subsystem])
	~	Defect
	~		|
Sum design faults generated per subsystem[subsystem]=
	SUM(Design faults generated[origin!,factor!,subsystem])
	~	Defect
	~		|

Sum design faults pending per subsystem[subsystem]=
	SUM(Design faults pending[origin!,factor!,subsystem])
	~	Defect
	~		|

Sum design faults undetected per subsystem[subsystem]=
	SUM(Design faults undetected[origin!,factor!,subsystem])
	~	Defect
	~		|

Sum code faults pending per module[module]=
	SUM(Code faults pending[origin!,factor!,module])
	~	Defect
	~		|

Code doc size[module]= INTEG (
	Code development activity[module]-Code verification activity[module]-Code not to verify\
		[module],
		0)
	~	KLOC
	~		|

********************************************************
	.Control
********************************************************~
		Simulation Control Parameters
	|
FINAL TIME  = 2000
        ~       Day
        ~       The final time for the simulation.
        |

INITIAL TIME  = 0
        ~       Day
        ~       The initial time for the simulation.
        |

SAVEPER  =
    TIME STEP
        ~       Day [0,?]
        ~       The frequency with which output is stored.
        |

TIME STEP  = 1
        ~       Day [0,?]
        ~       The time step for the simulation.
        |
```

APPENDIX B- SOURCE CODE OF THE GENSIM 2.0 WORKFORCE ALLOCATION FUNCTION

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_ac
tivities,num_devs);

      sum_assigned_skill[i]+=activity_skill(i,dev_cap_pattern,capabilities,dev
_skills,docs,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_ac
tivities,num_devs);

      sum_assigned_skill[i]+=activity_skill(i,dev_cap_pattern,capabilities,dev
_skills,docs,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_ac
tivities,num_devs);
      sum_assigned_skill[i]+=activity_skill(i,dev_cap_pattern,capabilities,dev
_skills,docs,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;
}
```

APPENDIX C- LIST OF ACRONYMS

CI	Code Inspection
CM	Configuration Management
DI	Design Inspection
DLL	Dynamic Linked Library
IT	Integration Test
IV&V	Independent Verification and Validation
RI	Requirement Inspection
SD	System Dynamics
ST	System Test
UT	Unit Test
V&V	Verification and Validation