Adaptation of a System Dynamics Model Template for Code Development and Testing to an Industrial Project

Berling, Tomas; Andersson, Carina; Höst, Martin; Nyberg, Christian

2003

Published in proceedings of ProSim 2003

Adaptation of a Simulation Model Template for Testing to an Industrial Project

Tomas Berling*, Carina Andersson**, Martin Höst**, and Christian Nyberg**

** Ericsson Microwave Systems AB*
SE-431 84 Mölndal, Sweden
Tel: +46 31 747 65 42
Fax: +46 31 747 04 84
e-mail: tomas.berling@emw.ericsson.se

*** Lund University*
Dept. of Communication Systems
Box 118, SE-221 00 Lund, Sweden
Tel: +46 46 222 33 19
Fax: +46 46 14 58 23
e-mail: (carina.andersson, martin.host, christian.nyberg)@telecom.lth.se

Abstract

Process understanding and improvements are essential in software industry in order to achieve cost effectiveness and short delivery times. One means of increasing process understanding and improvement is to utilize software process simulation.

This paper describes how a template model was created in order to increase the knowledge of the code development and test processes for an industrial organization. The template model was created from an existing system dynamics model for the unit test phase. The paper shows how the template model can be adapted and extended to fit a similar organization. The simulation model is applied for investigating the relationship between defect prevention in the development phase and defect detection in the various test phases. Data from a large contract-driven project were used in a case study to calibrate an adapted and extended model, which included code development and four test phases. Programmers and testers were involved in the design of the model.

The results show that it is possible to use the introduced template model and to adapt and extend it to a specific organization. We can also conclude that it is important to involve project members who contribute to the model building. The process understanding of the participating project members is increased due to their involvement.

1. Introduction

Simulation involves experimentation with a model of a system instead of the system itself. Usually the model of the system is implemented in a computer program. Some reasons for the increasing interest of using simulations in industry are:

- It might be dangerous to experiment with the system. If for example the system is a nuclear power plant, experimentation with a new control system is not allowed until it is simulated.
- The system might not exist. If for example a new aircraft is constructed, it is best to evaluate its performance using simulation before actually building it. It would be too expensive to build several different aircraft and measure their performance.
- Before changing an organization it is advisable to simulate the new organization to see if it meets the demands put on it.

The models used in simulation usually consist of a state description and a number of rules that describe how the state is changed with time, given a certain environment. The rules of change can be differential or difference equations.

Usually a distinction is made between discrete event simulation and continuous simulation [1]. In discrete event simulation the state of a system is changed only when certain events occur and is not changed between these events. A typical example is a queuing system where the state is the number of customers in the queue and the events are arrivals of customers and departures of customers. An example of a continuous simulation is when the air pressure around an aircraft is simulated as a function of time. Usually differential equations are used to describe state changes in models used for continuous simulation. It is also possible to combine discrete event simulation and continuous simulation, which is usually called hybrid simulation, see for example Donzelli et al [2], and Martin et al [3].

In software engineering the main reasons for using simulations of software processes are for the purpose of strategic management, planning, control and operational management, process improvement and technology adoption, understanding, and training and learning [4,5]. In a software development project the effect of a process change in the code development or the test phases can be difficult to predict or it can be difficult to prioritize work in the different phases during time pressure, for example.

A simulation model is appropriate to use in these cases. The risk of changing processes in the running projects in order to learn about it and to implement new ideas is too high, since it would lead to longer delivery times and high costs. A simulation model is used without any risk and with a relatively low cost.

The focus of this study is to enhance the modeling of the code development and test phases, for any organization, in order to understand the current software development process and to facilitate for future improvements to these processes. A system dynamics model with a code development phase and a test phase has been developed, which can be used as a template for other organizations to simulate these phases. The paper describes how this template model can be extended and adapted to suite the software development process in an organization.

The template model has been extended and adapted at Ericsson Microwave Systems AB, Sweden, to facilitate process improvements. Specifically the resources used, the distribution of undiscovered defects in the different test phases, and the cost of finding defects in different phases were studied.

The main research questions of this study are:

- What key tasks, primary objects, and vital resources, in the simplest case, are needed in a simulation model in order to investigate for example the resources used, the distribution of undiscovered defects in different phases, and the cost of finding defects in different phases?
- How can such a template model be adapted and extended to a specific organization?

The template model in this study is based on the study by Collofello et al [6], who modeled and simulated a unit test phase. The idea of viewing the unit test phase as two flows, a testing flow and a detection flow originates from Collofello et al, and in this study the model is further generalized.

Modeling and simulation of the code development and test phases have been performed in other studies. Analysis of the test process has for example been performed by Raffo et al [7] in which the impact of a process change was simulated. The process change involved the implementation of unit test plans and the simulation result showed that the process change would be successful. Madachy et al [8] have simulated the peer review model in an organization to investigate the dynamic project effects of performing inspections. The code development and test phases are parts of this model. The simulation results helped the planning and performance of peer reviews. Andersson et al [9] simulated the requirements specification and test phases and specifically analyzed the resource allocation in the different activities to decrease the project cycle time. The models used in these studies are specific for the examined organizations in contrast to the general model presented here.

In this paper a continuous simulation model is used. A discrete event simulation can also be used for this purpose. The discrete event simulation technique has for example been used to model a specific requirements management process for identification of overload situations [10].

The paper is structured as follows. The organization, developed products, and process are described in the environment part in Section 2. The method used is presented in Section 3 and the model and simulation is reported in Section 4. Conclusions are presented in Section 5.

2. Environment

2.1. Organization and Developed Products

The study is performed at Ericsson Microwave Systems AB, where radar systems are developed. The systems are large and complex with hard real-time constraints. The systems are divided into sub-systems, which are integrated at several levels, both hardware and software wise.

The products are delivered on contract. There are therefore relatively few customers compared to broad market products.

2.2. Process

The organization follows an incremental software development process. In each development step, called increment, functionality is added to the previous one. The functionality is added in a manner so that the system is always executable. The first increment contains only basic functionality and the last increment contains all functions.

In each increment the following development phases are included:

- System requirements specification
- Sub-system level 1 requirements specification (see Figure 2 for the different sub-system levels)
- Sub-system level 2 requirements specification
- Code development and unit test
- Sub-system level 2 verification
- Sub-system level 1 verification
- System integration
- System verification

System acceptance tests with the customer are performed after the last increment. Table 1 presents the development phases included in the case study and the personnel performing it.

Table 1. Development phases in the study and the personnel performing it.

Development phase	Personnel
Code development and unit test	Programmers
Sub-system level 2 verification	Programmers
Sub-system level 1 verification	Programmers
System integration	Independent testers
System verification	Independent testers

The sub-system level 1 requirements specification phase is performed by design engineers and the sub-system level 2 requirements specification phase is performed by programmers. These two phases are not included in the simulation study.

The sub-system is developed by approximately 4 programmers in average. The sub-system is divided into units, which are tested separately. The unit tests are developed and executed at the same time as the code development for the system. When the programmers have completed the code development and the unit tests are executed without failures the code is frozen in a unique revision and the next phase, sub-system level 2 verification, is performed. In sub-system level 1 verification, which is the next phase, the sub-systems at level 2 are integrated and verified into one sub-system at level 1. When this phase is completed the sub-system at level 1 is delivered to the independent test engineers. In the system integration phase the testers integrate the sub-system level 1 with several other sub-systems at level 1. When the integration phase is conducted the next phase, system verification, is performed. In the system verification phase the system is verified by the testers. When the system has been verified and defects have been corrected or postponed, the development of the increment has been completed.

Several increments can exist at the same time, but in different phases, i.e. the next increment can start before the previous is completed. Figure 1 shows an example of development phases and increments in an incremental development process. Figure 2 shows the sub-system level 2 in the study in relation to the whole system.

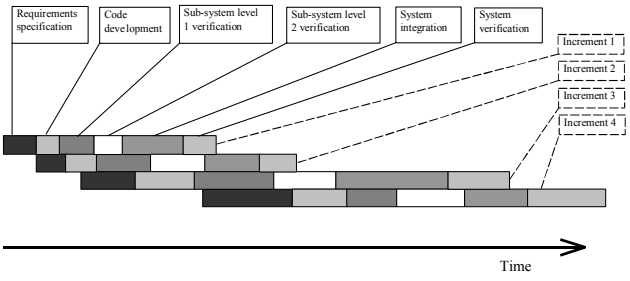

Figure 1. An example of development phases and increments in an incremental development process

The organization also follows a formal review process for all documents. All necessary documents are defined in the formal incremental software development process.

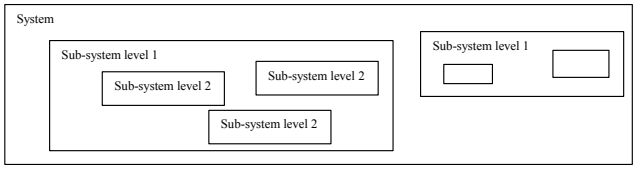

Figure 2. The sub-system level 2 in the study in relation to the whole system

3. Method

In order to answer the research questions, the idea of implementing a template model, and adapting and extending it to a specific organization is examined. Building the simulation model was an iterative procedure with a continuous contact with the programmers and testers in the modeled project. The close co-operation with the programmers and testers resulted in discussions on both model purpose, and model structure. The development procedure can be described in several steps, where feedback from the programmers and testers was received in every step.

The first step concerned specifying the purpose, model scope, result variables, process abstraction, and input parameters. This was performed according to a guideline of Kellner et al [5]. These aspects were identified in order to specify what to simulate.

The **purpose** of the simulation study is to enhance the understanding of the code development and testing phases, specifically the resources used, the distribution of undiscovered defects in the different test phases, and the cost of finding defects in different phases. When the understanding has increased the simulation model can be used for process improvement and technology adoption in the code development and test phases.

The **model scope** was confined to the development and testing phases. The requirements specification phases were excluded from the model's boundary for the reason that faults in the requirements specifications are only indirectly causing defects in the code, through the programmers' knowledge and skills. If the code would have been generated automatically from the requirements specifications the requirements specification phases would have been included. Even though the requirements specification phases are not unique parts of the model, they could be included as input parameters at the development, and testing phases.

The simulation of an industrial project was performed for one increment, see Figure 1, i.e. a portion of a life cycle, in one project.

The **result variables** in the project simulation included defect distribution between the phases, resources used in the phases, and an estimated cost of finding defects in different phases.

The **process abstraction** part is the key contribution of this paper. The main research questions presented in Section 1 yield the process abstraction. The key tasks, primary objects, and vital resources according to Table 2 were identified as the simplest case for the template model. The idea of viewing the unit test phase as two flows, a testing flow and a detection flow originates from Collofello et al [6]. The template model is built from this idea.

Table 2 Key tasks, primary objects, and vital resources for the template model.

Key tasks	Primary objects	Vital resources
Code development	Incoming work in KLOC	Programmers
Testing of code	Defects in code	Testers

This template model can be adapted and extended with further key tasks, primary objects, and vital resources to suite the industrial environment.

The key tasks, primary objects, and vital resources in the industrial simulation included in this study were adapted and extended according to Table 3. This adaptation and extension is directly related to the organization process, described in Section 2.2.

Table 3 Key tasks, primary objects, and vital resources for the industrial environment in this study.

Key tasks
Code development and unit test
Sub-system level 2 verification
Sub-system level 1 verification
System integration
System verification
Rework of defects, i.e. corrections.
Primary objects
Incoming work in KLOC
Defects in code
Vital resources
Programmers
Testers

The key tasks are the activities relevant to the model purpose, while the primary objects are the project artifacts, believed to affect the result variables. Vital resources could also be the hardware used for code development and testing, but this was not included in this industrial simulation.

A case study by Berling and Thelin [11] of the verification and validation activities in the organization served as a baseline for the important factors and the expected behavior of the simulated system. In their study, the trade-off between inspection and testing, in terms of faults found and resources used were investigated in the organization. Data from their study were used to calibrate and validate the adapted simulation model.

The **input parameters** are defined in accordance with the desired result variables and the process abstraction. In the template model the in-parameters were defined according to Table 4.

Table 4. The in-parameters in the template model.

Input parameter
Incoming work
Programmer Resource
Tester Resource
Coding Method
Testing Method

In the industrial simulation included in this study the template model was extended with further in-parameters. Most of the in-parameters in the industrial setting are constants, defined by the model user before the simulation model is executed, while others are varying over time. The input parameters are described in more detail in Appendix A.

With the simulation purpose, the model scope, the result variables, the key tasks, and the input parameters in mind the template model was adapted and extended to a first draft on paper. The draft model only consisted of qualitatively affecting relationships, and was without weighting and quantitative relationships.

To ensure the validity of the draft model, feedback was received from programmers and testers on the included in-parameters and relationships. Walkthroughs of the model were performed. Their comments mainly concerned definitions and effects of in-parameters and cost aspects of finding defects in different phases. Test coverage was for example one in-parameter added to the model after comments from the programmers and testers.

According to the programmers' and testers' comments the model was revised and thereafter transformed into the simulation tool. A visual description was chosen in order to enhance the understanding of the model, and to ease the calibration of the model, which continuously was performed with assistance from programmers and testers.

The development of the model extended from the template model, with few affecting factors, to a more detailed and project specific model with more relationships and inter-dependencies. As a result of the study by Berling and Thelin [11] the factor "Low-level design" was added to the model. This factor became apparent when faults found in the real system were classified and analyzed, i.e. faults were injected in the real system due to an inadequate low-level design for the sub-system.

In addition to the walkthroughs with programmers and testers, and using their estimates based on their past experiences, calibration of the simulation model was performed with real project data, see Section 4.2 for a description. If project data are not available statistical data

from literature can be used initially, see for example Jones [12].

The opportunity to further develop the model still exists, either to include or exclude activities, if these are assumed to affect the output, or to make changes to adapt the model for another development project.

4. Model and simulation

4.1. Template Model

The template model, including only the necessary key objects in the simplest case, is presented in Figure 3. This model consists of one module for code development, module A, and one module for test, module B. The arrow in Figure 3 corresponds to undiscovered defects, which are transferred from module A to module B. With the template model the user can simulate the number of injected defects during code development and the number of detected defects during testing as well as used resources and the time for development and testing. When the template model behavior is understood by the user, the model can be extended and adapted to reflect the industrial setting. This is described in the next section.

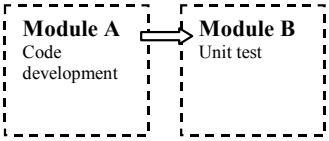

Figure 3. The template model with one development phase and one test phase.

The code development module, module A, is modeled according to Figure 4. The model user estimates the following input parameters:

- The incoming work
- The number of programmers
- The average number of injected defects per day per programmer
- The average produced number of KLOC per day per programmer with the coding method

The values of the input parameters can be estimated by project measures, reported statistics, or best estimates from experienced programmers and testers. The lower flow in Figure 4 corresponds to the coding rate, which is determined by the number of programmers and the average KLOC per day produced per programmer. The number of KLOC in incoming work together with the coding rate determine the number of days it takes to complete the code. The upper flow in Figure 4 corresponds to the defect injection rate during coding, which is determined by the coding rate and the injected number of defects per KLOC by the programmers, due to the coding method. The output from the module is the number of undiscovered defects in the code, which is transferred to undiscovered defects in module B, unit test, when module A has been completed. The formulas used in module A are:

- CodingRate=CodingMethodCR*ProgrammerResource
- Defect Injection Rate During Coding = Coding Rate * Coding Method DI

The Coding Method is divided into Coding Method CR for the coding rate, in which the unit is KLOC/day, and Coding Method DI for the defect injection rate, in which the unit is the number of defects/KLOC.

The input parameters in module A and their units are listed in Table 5.

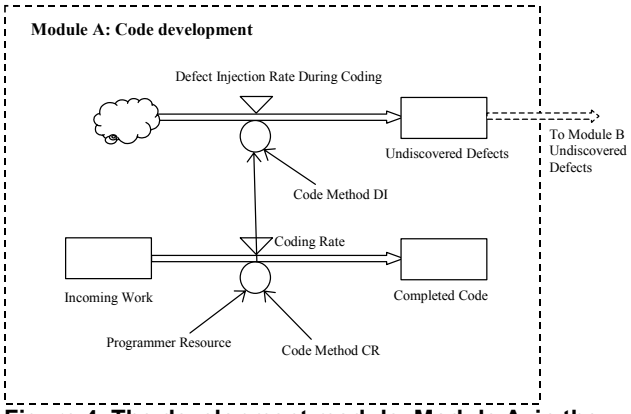

Figure 4. The development module, Module A, in the template model.

Table 5. Input parameters in Module A

Input parameter	unit
Incoming work	KLOC
Programmer Resource	Number of programmers
Coding Method CR	KLOC/day
Coding Method DI	Number of defects/KLOC

The test module, module B, is modeled according to Figure 5. The incoming work, the number of testers, the average number of detected defects per day per tester, and the average number of KLOC tested per day per tester with the test method is estimated by the model user. The lower flow in Figure 4 corresponds to the testing rate, which is determined by the number of testers and the average KLOC tested per day per tester. The number of KLOC in incoming work together with the testing rate determine the number of days it takes to test the code. The upper flow in Figure 4 corresponds to the defect detection rate, which is determined by the testing rate and the detected number of defects per KLOC with the test method. The output from the module is the number of undiscovered defects in the code. The formulas used in module B are:

- Test Rate = Test Method TR * Tester Resource
- Defect Detection Rate = Test Rate * Test Method DD

The Test Method is divided into Test Method TR for the testing rate, in which the unit is KLOC/day, and Test

Method DD for the defect detection rate, in which the unit is the number of defects/KLOC. The detection rate is independent of the number of faults in the code. This was chosen for practical reasons.

The input parameters in module B and the units are listed in Table 6. The number of tested KLOC per day is more difficult to estimate than for example the number of tested requirements per day. The unit number of tested KLOC per day is used anyway in order for the test method to be estimated in number of defects per KLOC in the upper flow. The unit in the upper flow would otherwise be the number of defects per requirement, which is also a difficult unit. A suggestion for the model user is to approximate that each requirement is of equal size in KLOC. A model extension with the input parameter test coverage, for example, can be performed by measuring test coverage by the number of requirements tested, and then approximating the corresponding number of KLOC tested. This approximation is used in this industrial simulation.

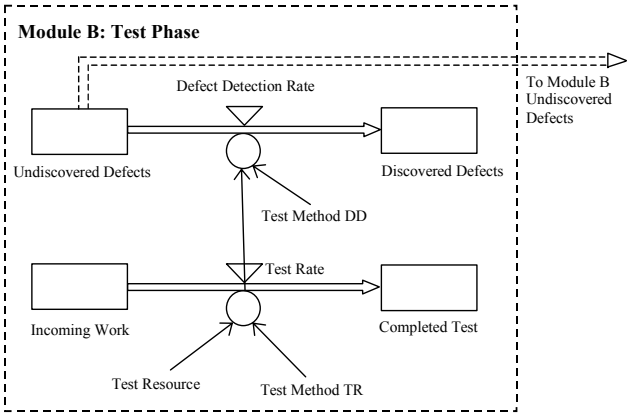

Figure 5. The test module, Module B, in the template model.

Table 6. Input parameters in Module B

Input parameter	unit
Incoming Work	KLOC
Test Resource	Number of testers
Test Method TR	KLOC/day
Test Method DD	Number of defects/KLOC

4.2. Simulation with an adapted model in an industrial setting

The template module described in Section 4.1 was implemented, extended and adapted in the organization described in Section 2. The in-parameters listed in Table A1 in Appendix A were considered important for module A. The major adaptations in module A are the inclusion of unit tests in the development phase and the extension of in-parameters to the code rate and defect injection rate, see Figure 6. The unit test is included in module A, since it is developed and executed in parallel with the code development in the same phase, see Section 2.

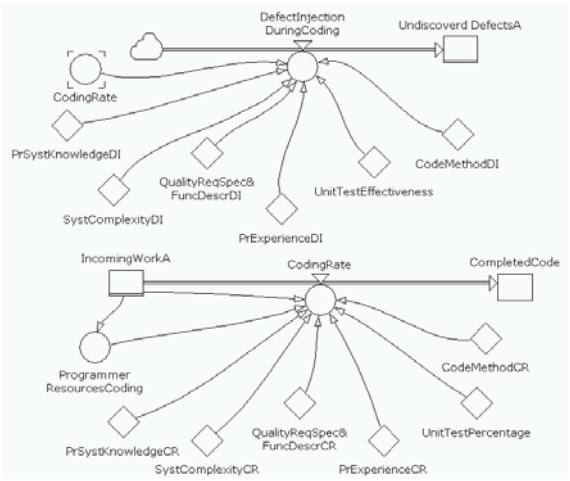

Figure 6. Module A in the extended and adapted model in the industrial setting.

The in-parameters listed in Table A2 in appendix A were considered important for module B. The major adaptations in module B from the template model are the extension of a rework flow and the extension of in-parameters to the test rate and defect detection rate, see Figure 7. The rework flow was added in order to estimate resources and time for the corrections of defects. Module B was also extended with a defect injection flow, due to the fact that new defects could be injected in the system during defect correction, see the flow from the cloud in the upper part of Figure 7. The arrow from Defect Rework Rate to Injected Defects due to rework is added in order to control the number of new injected defects, which is dependent on the number of corrected defects. The flow from Undiscovered Defects A to Undiscovered Defects B1 is added in order to transfer the Undiscovered Defects from module A to module B when module A is completed. The arrows from Undiscovered Defects A, Undiscovered Defects B1, and Incoming Work B1 to Test Resources B1 are added in order to start Module B when the Undiscovered Defects have been transferred from module A to module B. The arrow from Detected Defects B1 to PrResource Rework controls that programmer resources are only correcting defects if defects are discovered. The test coverage is controlled by multiplying the Incoming Work with the percentage of test coverage in module B.

The model was also extended with a modified module B for each testing phase, according to the software development process described in Section 2. The model for the industrial setting includes four modules of type B, according to Figure 8. The modules of type B are identical, but the parameters' values differ between the modules to re-

flect the situation in each phase. Defects from other subsystems at level 1 and 2 are not included in this study.

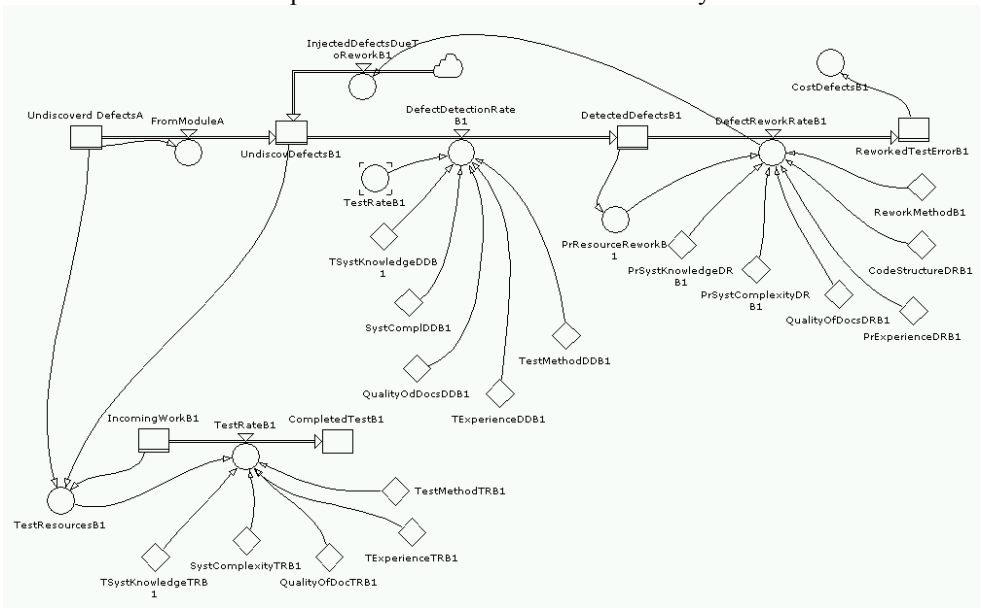

Figure 7. Module B in the extended and adapted model in an industrial setting.

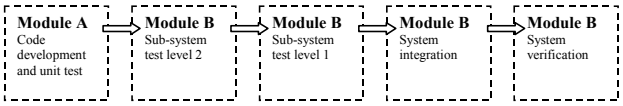

Figure 8. Adapted and extended model in an industrial setting.

The model in-parameters were adjusted to correspond to a real increment in a project with the programmers' and testers' viewpoint on the magnitude of the parameters. The data, which were used to calibrate the model, were taken from a problem reporting system, a personnel time logging system, and the number of lines of code from project data, as well as experiences from the programmers and testers. For more details on these data see [11]. The results of the simulation are presented in Figure 9.

The upper graph in Figure 9 shows the number of undiscovered defects in the different phases. The first steadily growing curve corresponds to the defect injection during coding and unit test. The decrease of the number of undiscovered defects in the test phase curves corresponds to the discovered defects during the test phases. The low increase of undiscovered defects in the test phase curves (see third description in upper part of Figure 9) is due to a low injection rate of new defects in the project during defect corrections. The verification phase in this increment was not performed. The undiscovered defects are therefore not reduced in the last curve, in this case. The model was calibrated to correspond to the real time scale and to the number of defects in the increment. The programmers and testers adjusted the in-parameters to simulate a process change to see how the model worked. The simulation results reflected the simulated change in the number of detected defects in the different phases.

The middle graph shows the number of persons, i.e. resources used in the different phases. The presented resources also include the programmers doing rework during defect corrections. The times in which the resources are zero are due to the model implementation. In this model the transfer of undiscovered defects from one module to another is completed before the next module testing is initiated. The model can be further developed in this respect.

The lower graph shows an estimate of the cost of detecting defects in the various phases. The cost of finding and correcting a defect, in the adapted model, is modeled to be increased for each test phase. This corresponds to the increased cost of performing all test phases again for the corrected defect. The actual flow or performance of new corrected releases of code, due to defect corrections, is not simulated in the model. The largest cost curve in the lower graph is due to the undiscovered defects, which were not found in this increment. The cost of finding defects in different phases is difficult to estimate, since each defect can cause different costs. Various approximations and definitions can be used. In this case the cost of finding faults in different phases were approximated with a fictitious value of 10 for the first testing phase, 20 for the next, and so on. The total cost, which is not shown in Figure 9, of all the found and not found defects in the different phases yields a good estimate for a process change in terms of costs.

4.3. Validation

The adapted model was validated with a sensitivity analysis [13]. In the sensitivity analysis the output variables Number of faults, and Calendar time were measured for

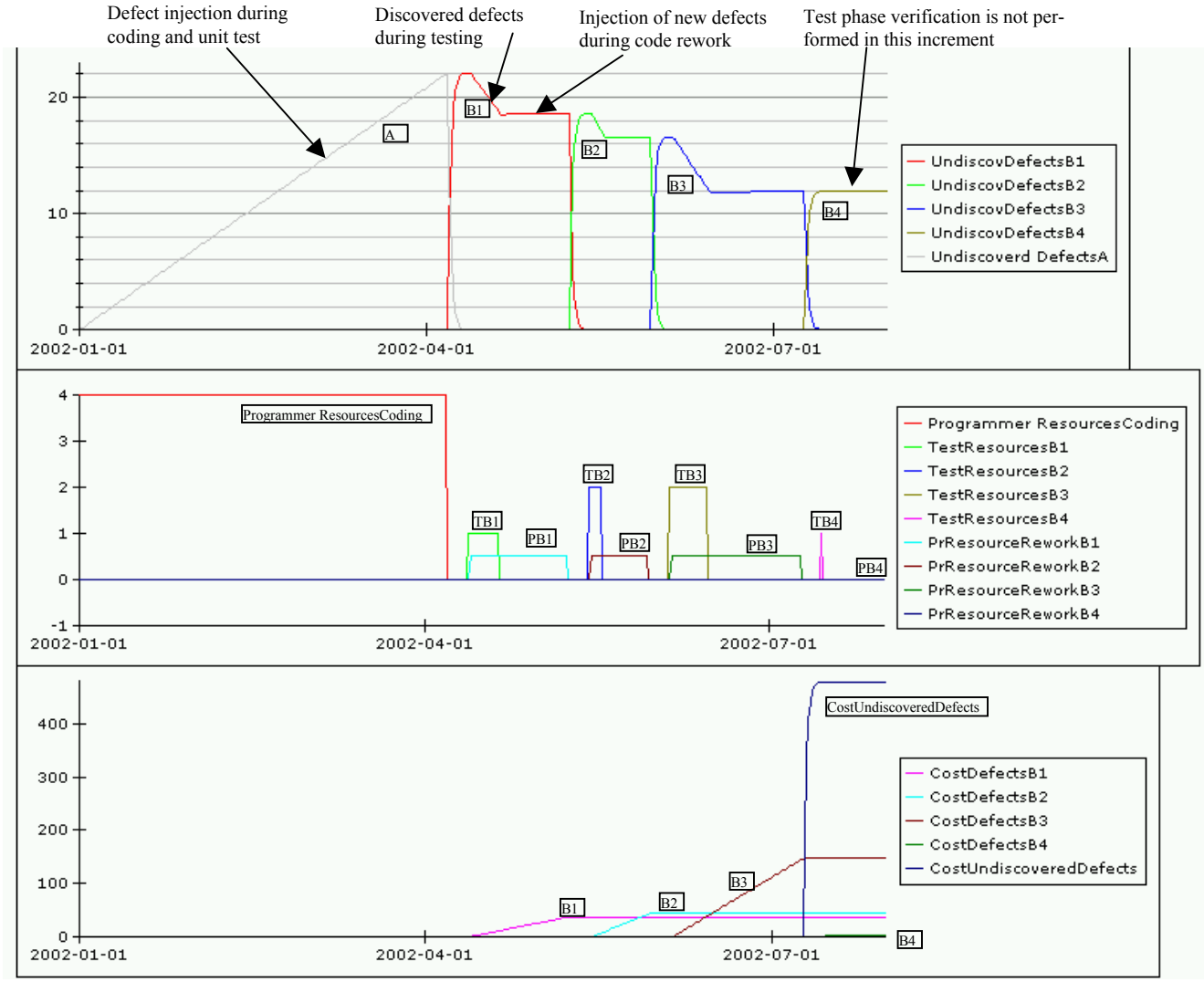

Figure 9. Results from the simulation model in the industrial setting

Module A when changing the input parameters Code Method Code Rate, Programmer Resource, Unit Test Percentage, Code Method Defect Injection, and Incoming work at "extreme values". The parameter Code Method Code Rate includes the parameters PrSystKnowledgeCR, SystComplexityCR, QualityReqSpec&FuncDescrCR, PrExperienceCR, and CodeMethodCR. The parameter Code Method Defect Injection includes the parameters PrSystKnowledgeDI, SystComplexityDI, QualityReqSpec&FuncDescrDI, PrExperienceDI, and CodeMethodDI. This simplification can be performed since these parameters technically are summarized into one parameter in the model.

The "extreme values" of the parameters were chosen by selecting a reasonably high and low value in a range for which the model is used. The number of programmers was for example 2 in the lower limit and 8 in the upper limit. The sensitivity analysis was performed with a full factorial design [14] for Module A, which results in 32 runs (5 parameters with 2 levels). The factorial design analysis showed that the number of injected faults are dependent on, and only on, the parameters Code Method Defect Injection, and Incoming work, and in fact Code Method Defect Injection*Incoming work. This is a correct behavior of the simulated system. The validation of the Calendar time showed that the Unit Test Percentage had been incorrectly implemented, since the calendar time increased when the unit test was reduced. The model was corrected and a validation was performed a second time with a correct behavior for all parameters.

The validation of Module B was performed with the parameters Test Method Test Rate, Test resource, Test

Method Defect Detection, Programmer Resource Rework, Incoming work, Rework Method, and Injected Defects due to rework. A simplification of the parameters Test Method Test Rate, Test Method Defect Detection, and Rework Method was performed, similarly as for Module A. The output variables were the number of detected faults, the number of days for rework, and the number of test days. These output variables are used for the calculation of costs etc. A fractional factorial design with 16 runs was performed. This means that first-order effects cannot be separated from third-order interactions, but effects of third-order interactions are not considered likely in this case, thus not affecting the result. When choosing the "extreme values" for Module B, certain relationships between parameters set limitations. For example the test rate (KLOC/day) could not be greater than the incoming work (KLOC) if the time step is set to 1 day. The analysis of the fractional factorial design showed that the model behaved correctly for all parameters and output variables.

5. Conclusions

In this study a template model has been developed and evaluated. The template model has been specialised into a model that is adapted to a specific industrial project. We have found that it is possible to use the template model when a specific model is derived, and that it is possible to derive the specialised model as it was done in the presented case study. We have also seen that it is important to involve representatives from the project. In the case that is presented, the representatives came from the project that was simulated, and we believe that this is a feasible way in cases where this is possible. The programmers and the testers had many important suggestions and corrections in the work with the specific model.

It is also concluded that a thorough analysis of project data, yielding information regarding resources used, faults found etc in the phases facilitate the model building and validation.

During the feedback-session it was found that the programmers and testers were interested and they thought that they had gained understanding of the process because of this work. We therefore believe that the model describes issues that are important, and that it is a good representation of the real process.

We believe that it is possible to use the template model in organisations that are similar to the studied organisation. It is probably possible to adapt the model in the same way as in this study, if the project does not differ very much.

Further work includes more experimentation with the template model. For example, the organisation in the case study is planning to use the adapted and extended model in more increments.

Acknowledgement

The authors would like to thank our colleagues at Ericsson Microwave Systems AB Maria Jonsson, Reine Larsson, Magnus Larsson, Carl-Ejnar Bergh, and Thomas Svensson for their contribution to this work.

This work was partly funded by The Swedish Agency for Innovation Systems (VINNOVA), under a grant for the Centre for Applied Software Research at Lund University (LUCAS).

References

[1] Law, A. M, and Kelton, W. D., *Simulation modeling and analysis,* 3rd ed., McGraw-Hill, 2000.

[2] Donzelli, P., Iazeolla, G., "Hybrid simulation modelling of the software process", *Journal of Systems and Software*, *Vol 59, Issue 3,* 2001, pp. 227-235.

[3] Martin, R., Raffo, D. "Application of a hybrid process simulation model to a software development project", *Journal of Systems and Software*, *Vol 59, Issue 3,* 2001, pp. 237-246.

[4] Abdel.Hamid, T., Madnick, S., *Software Project Dynamics: An Integrated Approach*, Englewood Cliffs, New Jersey, Prentice Hall, 1991.

[5] Kellner, M.I., Madachy, R.J., Raffo, D.M., "Software process simulation modeling: Why? What? How?" , *Journal of Systems and Software*, *Vol 46, Issue 2-3,* 1999, pp. 91-105.

[6] Collofello, J.S.; Zhen Yang; Tvedt, J.D.; Merrill, D.; Rus, I., "Modeling software testing processes", *Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications*, 1996, pp. 289-293

[7] Raffo, D. M., Kellner, M. I., "Analyzing Process Improvements Using the Process Tradeoff Analysis Method", *Proceedings of the Software Process Simulation Modeling Workshop (PROSIM 2000), Held in London, UK, July 12-14*, 2000.

[8] Madachy, R., Tarbet, D. "Case studies in software process modeling with system dynamics", *Software Process Improvement and Practice*, *Vol 5, Issue 2-3,* 2000, pp. 133-146.

[9] Andersson, C., Karlsson, L., Nedstam, J., Höst, M., Nilsson, B. I., "Understanding software processes through system dynamics simulation: a casestudy", *Proceedings of 9th Annual IEEE International Conference and Workshopon the Engineering of Computer-Based Systems (ECBS 2002). Held in Lund,Sweden, April 8-10*, 2002, pp. 41-48.

[10] Höst, M., Regnell, B., Natt och Dag, J., Nedstam, J., and Nyberg, C. "Exploring bottlenecks in market-driven requirements management processes with discrete event simulation", *Journal of Systems and Software*, *Vol 59, Issue 3,* 2001, pp. 323-332.

[11] Berling, T., Thelin, T. "An Indusrtial Case Study of the Verification and Validation Activities", *submitted to the Metrics Conference 2003.*

[12] Jones, T. C., Estimating Software Cost*,* McGraw-Hill, 1998

[13] Banks, J., Handbook of Simulation, John Wiley & Sons, 1998, ISBN 0-471-13403-1.

[14] Box, G. E. P., Hunter, W. G., and Hunter, J. S., *Statistics for experimenters: An introduction to Design, Data Analysis, and Model Building*, Wiley-Interscience, 1978, ISBN 0-471-09315-7.

Appendix A

Table A1. Important in-parameters for module A in the extended and adapted model.

Input parameters Module A	Measure
Programmers' system knowledge Code rate	Consider the characteristics in Table A3 below "Programmer participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the reduced or increased production of KLOC/day, due to level of system knowledge.
System complexity Code rate	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the reduced or increased production of KLOC/day, due to level of system complexity.
Quality of requirements specifications and functional descriptions Code rate	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the reduced or increased production of KLOC/day, due to level of quality of requirements specifications and functional descriptions.
Programmers' experience Code rate	The programmers' experience as a programmer and with the language used. Estimate a measure on the reduced or increased production of KLOC/day, due to level of programmers' experience
Coding method code rate	The number of produced KLOC/day. Estimate a measure of the number of produced KLOC/day per programmer, due to the coding method used.
Programmer resource for coding	The number of programmers for the sub-system development.
Amount of incoming work (KLOC)	Lines of uncommented code
Programmers' system knowledge Defect injection	Consider the characteristics in Table A3 below "Programmer participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of injected defects/KLOC, due to level of system knowledge.
System complexity Defect injection	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of injected defects/KLOC, due to level of system complexity.
Quality of requirements specifications and functional descriptions Defect injection	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of injected defects/KLOC, due to level of quality of requirements specifications and functional descriptions
Programmers' experience Defect injection	The programmers' experience as a programmer and with the language used. Estimate a measure on the number of injected defects/KLOC, due to level of programmers' experience.
Unit test effectiveness	The number of defects/KLOC discovered by unit test.
Coding method defect injection rate	The number of injected defects/KLOC. Estimate a measure of the number of injected defects/KLOC per programmer, due to the coding method used.

Table A2. Important in-parameters for module B in the extended and adapted model.

Input parameters Module B	Measure
Testers' system knowledge test rate	Consider the characteristics in Table A3 below "Tester participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of tested KLOC/day, due to level of system knowledge.
System complexity test rate	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of tested KLOC/day, due to level of system complexity.
Quality of requirements specifications and functional descriptions test rate	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of tested KLOC/day, due to level of quality of requirements specifications and functional descriptions.
Testers' experience test rate	The testers' experience. Estimate a measure on the number of tested KLOC/day, due to level of testers' experience.
Test method test rate	The number of tested KLOC/day. Estimate a measure of the number of tested KLOC/day per tester, due to the test method used.
Test resource	The number of testers in the test phase.
Incoming work	The number of KLOC of the sub-system to be tested .
Test coverage	The % of code tested. (Multiplied with incoming work in the model)
Testers' system knowledge defect detection	Consider the characteristics in Table A3 below "Tester participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of detected defects/KLOC, due to level of system knowledge.
System complexity defect detection	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of detected defects/KLOC, due to level of system complexity.
Quality of requirements specifications and functional descriptions defect detection	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of detected defects/KLOC, due to level of quality of requirements specifications and functional descriptions
Testers' experience defect detection	The testers' experience. Estimate a measure on the number of detected defects/KLOC, due to level of testers' experience.
Test method defect detection	The number of detected defects/KLOC. Estimate a measure of the number of detected defects/KLOC per tester, due to the test method used.
Programmers' system knowledge Defect rework	Consider the characteristics in Table below "Programmer participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of reworked defects/day, due to level of system knowledge.
System complexity Defect rework	Consider the characteristics in Table below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of reworked defects/day, due to level of system complexity.
Quality of requirements specifications and functional descriptions Defect rework	Consider the characteristics in Table below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of reworked defects/day, due to level of quality of requirements specifications and functional descriptions.
Programmers' experience Defect rework	The programmers' experience as a programmer and with the language used. Estimate a measure on the number of reworked defects/day, due to level of programmers' experience.
Code structure Defect rework	The degree to which the code is well-structured and well-documented. Estimate a measure on the number of reworked defects/day, due to level of code structure.
Programmer resource for rework	The number of programmers for the sub-system development. Since the rework is performed in a later phase it is not a conflict with the programmer resource for coding in the model. The measure should reflect the average number of programmers used for rework
Rework method rework rate	The number of reworked defects/day. Estimate a measure of the number of reworked defects/day per programmer, due to the rework method used.
Defect injection rates during code rework	Estimate the % of defects that lead to new defects.

Table A3. Characteristics for a number of in-parameters in the extended and adapted model.

Characteristics	Measure
Programmer participation in reviews	Important documents for code development is reviewed.
Number of years with total system	Number of years of work with total system, in order to know the purpose, structure etc. of the system.
Number of years with sub-system	Number of years of work with sub-system, in order to know the purpose, structure etc. of the sub-system.
Used system in laboratory environment	The programmer or tester has used the system in laboratory environment.
Common components	The use of common components affects the complexity
Sub-system's control	The sub-system's control and effect on other sub-system
Other sub-systems' control	The degree to which the sub-system is controlled and affected by other sub-systems.
Documentation status	The degree to which important documents (requirements specifications and functional descriptions) are complete, i.e. if important parts are missing.
Review of documents	The amount of review and the appropriateness of reviewers.
Faults in documents	The degree of faults found in important documents (requirements specifications and functional descriptions) after release.
Tester participation in reviews	Important documents for testing is reviewed.