EVALUATION OF SOFTWARE QUALITY ATTRIBUTES DURING SOFTWARE DESIGN

C. Wohlin
Dept. of Communication Systems, Lund Institute of Technology, Lund University, Box 118, S-221 00 Lund, Sweden

Keywords: quality evaluation, performance, functional analysis, software reliability, simulation, software metrics, Cleanroom

Edited by: M. Kubat
Received: October 28, 1993 **Revised:** January 18, 1994 **Accepted:** February 28, 1994

This article presents an evaluation method of software quality attributes during software design with a high-level design technique. The attributes considered are real time functional behaviour, performance (in terms of capacity) and reliability. The method is based on transformation of the software design documents and simulation models of hardware architecture in terms of processors, communication channels etc. and the environment in terms of usage of the system. The method provides an opportunity to concentrate on software, architecture and usage of the system one by one and facilitates analysis of the software system long before it is taken into operation, which is particularly valuable for safety-critical software and other complex software systems. This implies that important information concerning both functionality, performance and reliability can be studied early in the development, so that re-design can be performed instead of implementing a poor solution. These early insights become more and more

1 Introduction

The quality of the software in the operational phase is a critical issue. The cost for software failures in the operational phase of software systems today is very high, both in risks and economical terms. This is in particular the case for safety-critical software and other large software systems where the society depends heavily on the software.

The software community is not in control of the quality of the software. Cleanroom Software Engineering [5], [6] and [10] is a promising attempt in dealing with these problems, but it is not enough. Methods are needed for early analysis of functionality, performance as well as reliability.

The dependability of software systems is highly due to getting things right from the beginning. This is essential since the main principles of a software system are often built into the product at an early stage of the software life cycle. The inevitable conclusion from this is that it is necessary to have methods for modelling and analysing the behaviour of the software long before it goes into operation.

A method for evaluation of performance, reliability and functional aspects of software systems at an early stage is presented. The method supports prediction of important quality attributes at the software design phase. The evaluation and prediction is based on the actual software design and simulation models of architecture and user behaviour.

The article presents initially some objectives of the work before discussing some related work. The concepts in the method are then described from a general point of view. The general concepts are exemplified with SDL (Specification and Description Language, standardized by ITU, [3] and [4]), i.e. transformation rules are presented and an example is described to show the applicability of the method. Finally some general conclusions are presented.

2 Objectives

The main objective of this work is to formulate a general (independent of software description technique) method for functional, performance and reliability evaluation at an early stage of software development. The long term objective is to formulate a method that can be applied throughout the software life cycle to evaluate and assess the quality attributes of software systems. The objective is that the principles presented can be used throughout the software life cycle even if the actual level of detail in the models used may vary depending on available information.

The aim is to provide a method for evaluation of functional real time behaviour, performance (in terms of capacity) and reliability of software design descriptions. The method is based on that the software design descriptions are specified with a well-defined language, for example SDL, which can be transformed automatically into a simulation model of the software design. A tool prototype performing the transformations of SDL descriptions into a simulation model of the software has been implemented, [18] and [8]. It must be stressed that SDL is used as an example assuming that SDL is the normal software development method at the company applying the proposed method.

Transformation rules have been formulated for SDL, hence showing that it is possible to actually use the design in the evaluation of quality attributes instead of formulating a separate simulation model of the behaviour of the software. The transformed model is then distributed on a simulation model of the architecture. The input to the system (transformed software design distributed on a simulation model of the architecture) is then modelled in a usage model, which is a simulation model of the anticipated usage of the system. The method consists hence of three separate models: software model, architecture model and usage model.

The software model is a direct transformation of the actual design of the software to be used in the final system. The usage model and the architecture model are formulated in the same language as used in the software design, but these two models are supposed to be simulation models of the actual architecture and of the anticipated behaviour of the users of the system. The three models are hence described with the same description technique which is the same technique as the software is being designed in. The strength of the method is its opportunity to combine the actual software design with simulation models.

The usage model is used as a traffic generator to the system, i.e. it sends signals to the system in a similar way as expected when the system is put into operation. The reliability of the software can be evaluated since failures occur as they would in operation, since the usage model operates with a usage profile which describes the anticipated usage in terms of expected events. The capacity of the system is determined based on the inputs coming into the system and measurements on loads and throughputs. The analysis allows for analysis of bottlenecks in the system as well as delays. The real time functional behaviour is analysed in terms of locating unexpected functional behaviour. In particular, it is possible to find functional behaviour that is a direct consequence of the delays in the system.

The difference between the work presented here and other approaches is the opportunity to combine the software design with simulation models described in the same description technique as the software design. The idea in itself is general and no direct limitations concerning for which design techniques this approach can be applied have been identified. The objective has neither been to formulate a tool set nor to advocate the use of SDL. The major difference with existing approaches is that a special notation has not been used and hence the method is believed to be general and the method aims at more than one quality attribute. This implies that it should be possible to adapt the general idea and formulate transformation rules etc. for other design techniques as well. The aim is to provide a framework and a method supporting early evaluation based on the actual software design as well as other description levels in the future.

The advantages with the proposed scheme can be summarized by:

- the evaluation of quality attributes can be performed at an early stage, i.e. during the design (cf. below with for example statistical usage testing),
- the concepts are general even though transformation rules have to be formulated for

each specific design language,

- the actual software design is included in the evaluation method hence allowing for a good basis for decisions regarding the quality of the software,
- the method aims at analysing performance, reliability and real time functional behaviour hence no separate analysis has to be performed for each quality attribute.

3 Relation to other work

Some approaches resembling the functional and the performance evaluation can be found, see for example [1], [7], [14] and [15]. These methods do neither support reliability evaluation of the software nor do they base the analysis on the actual software design. One of the major differences with the approach presented in this article is its generality. The objective has been to clearly separate method from tools. The approaches presented in [1], [7], [14] and [15] primarily aims at using a "home made" notation for the problem and then building a tool supporting the proposed notation and method. The method presented here takes a more general approach by introducing concepts and ideas, which can be adapted to the design method and tools used in the ordinary software development process. The objective is to make the method independent from any specific description technique and tool environment.

Statistical certification of software, including usage modelling, is currently an area evolving rapidly. Statistical usage testing as a method for certification was initially proposed within the Cleanroom method [5], [6] and [10]. A similar approach is currently used and being developed at AT & T. It is concluded from the projects at AT & T, [9], [11] and [12], that the cost for system test and the overall project cost are reduced considerably. In [11], it is stated that the cost reduction for system test for a "typical" project is 56%, which is 11.5% of the total project cost. Usage modelling from Markov chains are discussed in [13] and [16]. The opportunity to perform early software reliability estimation from high-level design documents using dynamic analysis has been presented in [19]. The reliability certification is normally applied during the test phase, with exception of the work presented in

4 Overview: Modelling concepts

Three models are defined to formalise the modelling and evaluation process. The mapping between reality and models are depicted in figure 1. The models are denoted:

- Software Model
- Usage Model
- Architecture Model

The models, which will be explained below, are independent in the sense that the Software Model, the Usage Model and the Architecture Model are derived independently and they can be combined into an Evaluation/Simulation Model. The simulation is foremost intended to be used during the software design.

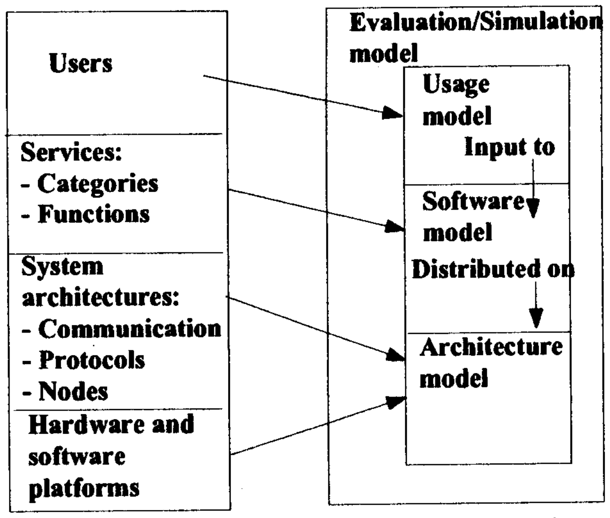

Figure 1: Mapping of the layers of users and system on the modelling concepts.

The Software Model and the Architecture Model are linked to each other through the distribution of the software units in the architecture. This means that the Software Model is allocated to the Architecture Model in a way that describes the actual distribution of the software in the architecture. The Usage Model generates the input to the simulated system (Software Model allocated on the Architecture Model). Thus the three models are connected together into the Evaluation/Simulation Model.

5 Model description

The models are derived in the following manner:

- Software Model
  The software descriptions (specification or design) are transformed to include the real time aspect of the software, which normally is not included in the software design. As part of the transformation the user is requested to add time consumption for executing different concepts of the software design. This addition of time consumption must be made based on prior knowledge or knowledge of the current system. The Software Model describes the application software, i.e. the services that the system provides to the user.

The content of the Software Model is shown in figure 2.

Software Model:
- Services and functions
- Logical flow
- Relative service times

Figure 2: Content of the Software Model.

- Usage Model
  The external usage is not normally described unless a reliability certification is to be made or as input to a performance simulation. The Usage Model must describe the structural usage of the analysis object (which refers to the part of the system being evaluated) and it shall be complemented with a usage profile to allow for reliability certification. The structural usage describes the behaviour of the user without quantification of the usage. The latter is added with the usage profile. Usage modelling with Markov chains is discussed in [13] and [16]. It is, however, favourable if the Usage Model is formulated in the same description technique as used in the software design. This is clearly possible as shown in [17] and in the example below.

The content of the usage model is shown in figure 3.

Usage Model:
- Analysis object
- User categories
- Usage states
- Input probabilities/ intensities
- Service demand

Figure 3: Content of the Usage Model

- Architecture Model
  The architecture is described in terms of a performance simulation model. This type of model is quite common and it is used extensively when doing performance analysis. The aspects to find are those governing the performance behaviour of the architecture. The objective here is to define a performance model of the architecture in the same description technique as has been used in the software design. This is also further discussed in [17] and in the example below.
  The content of the architecture model is shown in figure 4.

Architecture Model:
- Topology
- Interconnection devices
- Resources and servers
- Operating system features
- Algorithms (e.g. scheduling and routing)
- Flow-paths
- Capacity

Figure 4: Content of the Architecture Model.

The concepts of the method have hence been described and to illustrate the application of the method SDL is used. First a brief introduction to SDL is given, secondly the transformation rules for SDL are described and finally an example using the method is presented.

6 Brief introduction to SDL

The ITU Specification and Description Language, [2], [3] and [4], known as SDL, was first defined in 1976. It has been extended and reorganised in four study periods since this first definition. These have resulted in new recommendations for the language published every fourth year.

SDL is intended to be well-suited for all systems whose behaviour can be effectively modelled by extended finite-state-machines and where the focus is to be placed especially on interaction aspects. SDL is a unique language which has two different forms, both based on the same semantic model. One is called SDL/GR (graphical representation) and is based on a set of standardized graphical symbols. The other is called SDL/PR (phrase representation) and is based on program-like statements.

The main concepts in SDL are system, blocks, channels, processes and signals. These concepts form the basis for SDL, where system, blocks and channels describe the static structure while the dynamic behaviour is modelled with the processes and its signals. The processes are described by several symbols.

System: Each system is composed of a number of blocks connected by channels. Each block in the system is independent from every other block. Each block may contain one or more processes which describe the behaviour of the block. The only way of communication between processes in two different blocks is by sending signals that are transported by channels. The criteria leading to a certain division of the system into blocks may be to define parts of a manageable size, to create a correspondence with actual software/hardware division, to follow natural functional subdivisions, to minimize interactions, and others.

Block: Within a block, processes can communicate with one another either by signals or shared values. Thus the block provides not only a convenient mechanism for grouping processes, but also, a boundary for the visibility of data. For this reason, care should be taken when defining blocks to ensure that the grouping of processes within a block is a reasonable functional grouping. In most cases it is useful to break the system (or block) into functional units first and then define the processes that go into the block.

Channel: Channels are the communication medium between different blocks of the system or between blocks and the environment.

Signal: Signals can be defined at system level, block level, or in the internal part of a process definition. Signals defined at system level represent signals interchanged with the environment and between system blocks. Signals defined at block level represent signals interchanged between processes of the same block. Signals defined within a process definition can be interchanged between instances of the same process type or between services in the process. Signals are sent along signal routes between processes and on channels between blocks or when interchanged with the environment.

Process: A process is an extended finite-state-machine which defines the dynamic behaviour of a system. The extended finite-state-machine handles data within tasks and decisions. Processes are basically in a state awaiting signals. When a signal is received, the process responds by performing the specific actions that are specified for each type of signal that the process can receive. Processes contain many different states to allow the process to perform different actions when a signal is received. These states provide the memory of the actions that have occurred previously. After all the actions associated with the receipt of a particular signal have been performed, the next state is entered and the process waits for another signal. The basic concepts within a process are further described below.

Processes can either be created at the time the system is created or they can be created as a result of a create request from another process. In addition, processes can live forever or they can stop by performing a stop action. A process definition represents the specification of a type of process; several instances of the same type may be created and exists at the same time; they can execute independently and concurrently.

7 Transformation rules for SDL

7.1 SDL concepts

Each SDL concept or symbol must be associated with parameters that describe its behaviour in terms of performance, to create a Software Model that together with the Architecture Model and the Usage Model shall be put together to the simulation model. Initially the method has been focused on the most common used concepts, denoted Basic concepts (process level), which describes the behaviour within a process.

The parameters that can be associated with the symbols are:

- time, i.e. a delay in terms of an execution time (random or constant),
- probability, i.e. the probability for different outcomes when passing the symbol during the execution,
- signal reception or sending (intensities). The reception of a signal is based on the execution in another part of the analysis object or the environment, while a signal sending is based on the actual execution of the analysis object. This means that no intensity has to be directly associated with the symbol, but the symbol must be associated with a signal reception or sending all the same. The intensity is a consequence of the execution. The arrival intensities for signals arriving from the environment of the analysis object is part of the Usage Model, which models the surrounding of the analysis object.
- dynamic behaviour, i.e. creation or deletion of a process.

An estimate of one of these parameters can either be only one estimate or a combination of several estimates that have been weighed together. The basic concepts are associated with the parameters as follows:

Basic concepts (process level):

State - execution times, i.e. the times it takes to leave and enter a state respectively. Each transaction results in one time leaving a state and one time for entering a state, except when the transaction means that the process terminates its execution.

Input - signal reception and execution time, i.e. the time it takes to remove the signal from the queue and start the execution.

Output - execution time and signal sending. The execution time shall correspond to the time it takes to send the signal, while the signal sending is a direct consequence of reaching the output symbol.

Save - no parameters are associated with the save symbol, since it is assumed that the handling of the queue of a process is carried out by the processor responsible for that specific software process. This means that any delays caused by the queueing discipline shall be modelled by the Architecture Model. The save symbol has to be present after transformation as well to be able to save the right signals in the queue.

Task - execution time, i.e. the time it takes to perform the actions specified in the task.

Decision - execution time and probability. The execution time must correspond to the time it takes to evaluate the statement in the decision symbol and to choose execution path based on the evaluation. In cases when the criterion within the decision box can not be evaluated based on the specification, i.e. the symbol does only contain informal text, a probability for different paths is needed. The probability or probabilities shall model the number of times a certain execution path is chosen, based on the evaluation, compared to the total number of times the symbol is passed. The number of probabilities will be one less than the number of possible paths to leave the decision symbol, since the sum of the probabilities are equal to one.

Create request - execution time and dynamic creation. The execution time models the time it takes to create a new process and initiate all its values. The dynamic creation must be a part of the Software Model, since it is a vital part of the dynamic behaviour of the analysis object. This means that the symbol cannot just be translated to a delay.

Terminate process - execution time and dynamic deletion. The explanation is similar to the one about "create request", with the difference that a process is terminated instead of created.

Timer - Signal sending. The set and the reset command respectively are carried out within a task, which means that the execution time is associated with the task concept and not the timer concept. But the timer also means that a signal is sent to the process itself, and this signal sending has to be modelled in the Software Model. The internal signal sending is performed with the set concept.

Procedure - Execution time. The procedure symbols (call and return) themselves are only associated with execution times, while the result of the procedure call, i.e. the execution of the procedure, is modelled symbol by symbol within the procedure and according to the rules for respective symbol. This means that the procedure call symbol cannot be translated merely to a delay. In other words the procedure call symbol must remain in the Software Model.

Macro - The macro symbol is substituted with the content of the macro before the execution, which means that it can be disregarded from a performance point of view.

Join and label - Join and label have two main functions, either showing that the flow description continues on another page (in the description) or describing a "goto" in the program. The latter means that a jump is made to another part of the program. Instead of handling these two functions separately, it is assumed that the jump is done instantaneously independent if the jump is done to the next line or to another part of the program. This means that join and label are assumed to have no influence on the performance.

Asterisk - The asterisks are only a shorthand to be used as a wildcard, for example in a save symbol, where it means that all signals not explicitly received shall be saved. This shorthand is merely a simplification when working with SDL and it does not influence the behaviour of the actual implementation.

This discussion, of which performance parameters that have to be associated with the SDL symbols, leads to that transformation rules for each symbol can be formulated.

7.2 Example of some transformation rules

The transformation means that the system design in SDL is transformed into SDL descriptions describing the system from a performance perspective. Based on the parameters identified for each symbol transformation rules can be found. The transformation rules below include in some places symbols that are optional depending on the actual behaviour of the original description. A major objective for all concepts is that, if it is possible to put two or more transformations together it shall be done. For example, the execution times for two symbols after each other shall be put together to one delay. It must also be noted that the whole delay for a symbol is always assumed to be done first, before the actual event described by the symbol occurs. The delay can either be given a specific value or a random time, with some mean value, from some distribution. This means that when the task (set timer) is referred, it shall be possible to incorporate a random or constant delay if wanted. The assignment of values and the c

Basic concepts (process level): The basic concepts describe the dynamic behaviour of a process and they are translated to basic concepts describing their behaviour from a performance perspective. Most of the concepts, as described above, mean some sort of execution time (i.e. delay). The execution time can always be modelled with a task (set timer) and a state in which the process stays until the timer signal is received. Instead of describing this for each concept below a new meta-concept, which will be denoted "delay", will be used.

Definition of *Delay*:

Task (set timer): The timer is set to the time it takes to execute a particular symbol.

State: The "symbol" remains in this state until the timer signal is received.

The delay concept will be a shorthand for a task where a timer is set and a state where the process is waiting until the timer signal is received. The delay concept also includes the possibility of generating a random number corresponding to the execution time. The random number is drawn from a distribution with parameters that model the behaviour of the execution of the symbol. Some examples of how the basic concepts are transformed are:

State

Delay: The state symbol is really two different events that is leaving and entering a state respectively. The entering event is often referred to as "nextstate" from the perspective of the state we are leaving. This means that the delay can be divided into two parts, i.e. one delay for leaving the state and one delay for entering a new state (perhaps the same).

Input

Input: The signal must be received, since it can influence the forthcoming behaviour of the analysis object. *Delay*: The time to take the signal from the queue and to evaluate what to do has to be modelled as a delay.

Save

The save symbol will remain unchanged since it is needed to be able to handle the queue in an appropriate way, i.e. to save the right signals depending on the state of the process. It must be observed that the process is assumed to be in the last state until it reaches the next state and the states referred to are the "real" states of the original process and not the states that are introduced to cope with the delays. The reason for this is that the queue and the arriving signals shall be handled in an appropriate way, i.e. the signals shall be saved according to the actual system description.

Decision

Delay: The time to evaluate the statement within the decision symbol is modelled with the delay.

Task: If the decision symbol contains informal text a task is needed. The task shall be used to draw a random number from a uniform distribution to compare with the given probabilities for the different execution paths, which model the behaviour of the decision box in cases when the decision can not be evaluated from the original description.

Decision: The decision symbol is either left without any changes or in cases when the symbol contains informal text complemented with a decision criterion. The complement means that a random number is compared with the probabilities for different execution paths and a path is chosen according to the evaluation.

Task

Delay: The time to execute the task is modelled as a delay.

Create request

Delay: This delay models the time it takes to create a new process.

Create request: The new process is created in the SDL description describing the performance aspect as well.

Rules have in a similar way been formulated for all other basic concepts in SDL. The objective is to do the transformation automatically and in a dialogue with the user. The dialogue gives the user possibility to assign values to execution times, probabilities etc. The transformation rules and the concepts are evaluated through an example subsequently.

8 Example

8.1 Introduction

The objective of the example is to provide an illustration of how the proposed method may work and give a flavour of the type of analysis that can be made. This is made by going through some results from the example. It is impossible to go through the example in detail, but more information can be found in [17]. The example has been formulated using SDL (Specification and Description Language), [2], [3] and [4], but any other well-defined design technique could have been used. The example will show results concerning functional behaviour, performance and reliability.

The example describes the communication between two very simple telephone exchanges, which only provide the subscriber with the possibility to call a local call (within the same exchange) or long distance call (to the other exchange). The architecture is modelled with three SDL process types, one describing an exchange, one modelling the communication channel between the exchanges and one handling the administration of the architecture. The services provided by the exchange are designed with seven SDL process types. The usage is modelled with five SDL process types. Some of the process types are created and terminated dynamically, and several instances of the same process type may exist simultaneously. This means that the example in total includes 15 process types.

The system layout of the example including the environment is illustrated in figure 5.

The example can be summarised by:

- 5 processes modelling the behaviour of the subscribers.
  - *A_Subscriber*: models the behaviour of a phoning subscriber.
  - *B_Subscriber*: models the behaviour of a phoned subscriber.
  - *Call_Generator*: responsible for creating *A_Subscribers* as a new call shall start.
  - *Monitor*: responsible for creating *B_Subscribers* when connecting a call.
  - *Creator*: responsible for creating *Monitor* and *Call_Generator*.
- 3 processes modelling the architecture.
  - *Processor*: models the behaviour of the processors executing the software.
  - *Com_Link*: models the communication link between processors.
  - *Arch_Creator*: responsible for creating *Processors* and *Com_Links* according to the layout of the architecture.
- 7 processes designing the services provided by the software.

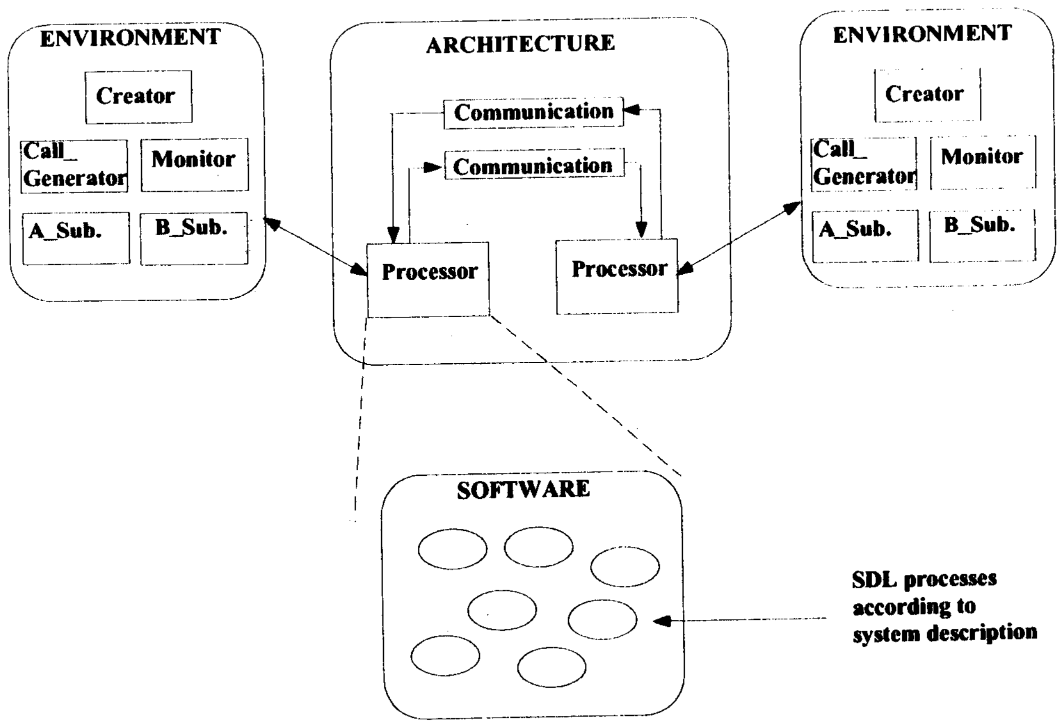

Figure 5: Layout of the example.

- *Statistics*: models the times when statistics about the processors (exchanges) shall be written on a file.
- *A_Handler*: handles the phoning subscribers.
- *B_Handler*: handles the phoned subscribers.
- *Digit_Handler*: responsible for the reception of digits and reserves a code receiver through communication with *Code_Receiver*.
- *Code_Receiver*: responsible for holding code receivers for on-going calls.
- *Soft_Monitor*: responsible for monitoring on-going calls at a processor.
- *Soft_Creator*: responsible for creating the *Soft_Monitors* necessary due to the architecture.

8.2 Software design in SDL

The SDL system description consists, as pointed out, of 7 processes, but before these are discussed in some more detail, the system and block level have to be discussed. The system consists of two blocks. The first block handles all activities that concern subscribers and the second one is a block responsible for collecting the statistics of the telephone exchange. The layout of the SDL design is shown in figure 6 and the processes and their communication are briefly described below.

The statistics block is simple, it only consists of the *Statistics* process of which one instance is created at the system start and it exists the whole life time of the system. The only thing to be noted with the process is that it calls a procedure regularly, which describes the times the statistics are put on a file.

The subscriber block consists of six processes, where one process is created by the system (*Soft_Creator*). This process is responsible for creating two monitors (one for each processor). The *Soft_Monitor* process is the receiver of incoming calls and it creates other processes that handles the subscribers, both A- and B-subscribers. The *B_Handler* process is created by the monitor in cases of a long distance call, otherwise the *B_Handler* process is created by the *A_Handler* process. The *B_Handler* process is quite easy and it handles the communication with the B-subscriber in the environment. The monitor process also creates the two processes controlling code receiving.

The *A_Handler* process creates a *Digit_Handler* process and is responsible for keeping the contact with the A-subscriber in the environment The

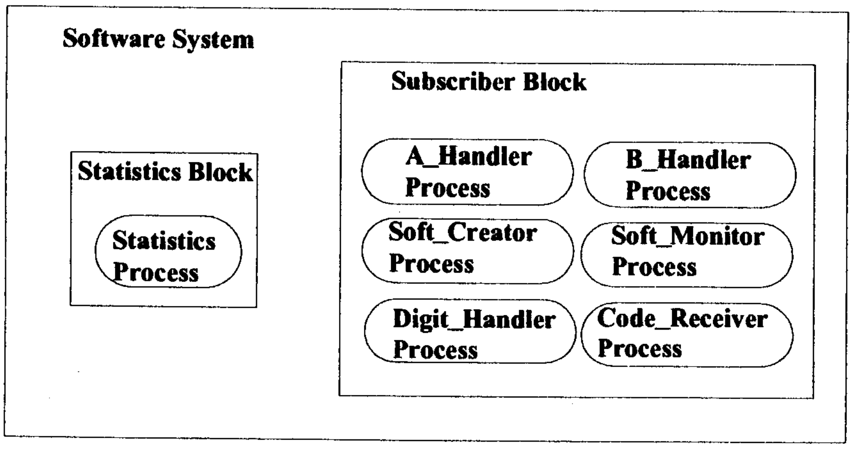

Figure 6: Software processes in the design.

Digit_Handler process checks if there are any code receivers free and if there are any it reserves a code receiver for the incoming call. The call is blocked if the code receivers all are occupied. The *Digit_Handler* is responsible for releasing the code receiver when it has been used.

The *Code_Receiver* process is not modelled in any detail at all. It consists mainly of signal receiving, signal sending and informal text. This is an important aspect, i.e. that the processes may be quite unspecified but still the quality attributes can be evaluated by applying this concept.

The tool support used allows for syntax and semantic analysis of the SDL system and code generation (SDL to C). The generated code was then compiled and linked. The tool allows for functional simulation without taking real time into consideration. A functional simulation was performed but it failed since the SDL system was not complete, i.e. some of the behaviour was not formally specified, it was only given in informal text. The problem with the informal text is in particular decision boxes with informal text since this means that the actual execution is not formally described. The decision boxes with informal text are translated when doing the transformation into a the Software Model.

8.3 Identify objectives of simulation

The objective of the simulation shall not influence the translation technique applied, but it may influence the way the architecture and the usage are described in the model. The objectives of the simulation can lead to that a measurement process has to be included in the simulation system. In this case the objectives are:

- Determine the total execution times of each software process type on the processors. This provides a possibility to identify the software bottleneck, since the available execution time is known through the simulated time. This must not be mixed up with the execution time of the simulation program.
- The load on the communication links is measured.
- Identify any functional problems.
- Detect faults that influence the perceived reliability of the software.

These objectives affect the process modelling the processor, the process modelling the communication link and the Software Model processes. A complement is needed in the Software Model processes to be able to measure for each process type, see below. The latter measurement will require a separate measurement process. These objectives have to be modelled by the user, when describing the architecture, the usage and introducing a measurement process.

8.4 Software Model

The original SDL system is transformed applying the rules formulated for SDL. The transformation results in a new system level, where the original SDL system is a block. This corresponds to the Software Model discussed above. The new system level also contains three new blocks, i.e. one block modelling the architecture, one block modelling the usage and one general block (used to control the simulation and to make measurements). These three blocks are left without content. These blocks will be complemented with the Architecture Model and the Usage Model as well as general processes governing the simulation. These models have to be formulated by the user, see below. The Software Model and the generated empty blocks are illustrated in figure 7.

All block levels in the original system have been moved one step down in the hierarchical structure. The processes in the Software Model are the result of applying the transformation rules. It must in particular be observed that all symbols with informal text have been removed or replaced, but they are modelled in terms of delays.

As an example of a transformation we will consider the transformation of a task in an SDL process, see figure 8.

The task remains after the transformation if it contains a functional behaviour in which case it influences the outcome of the execution of the process. Independently of the content of the task, a procedure call is added before the execution of the task. This procedure is denoted the delay procedure and one parameter is passed to the procedure, i.e. the delay for the task. The length of the delay has to be determined by the user of the method, as a first approach every symbol of the same type is assigned the same delay. The procedure is also shown in figure 8. The procedure delays the execution for the specified delay by use of the timer concept in SDL. It must be noted that all signals are saved within the procedure, and the reason is of course that the transformation may not alter the original functional behaviour. The transformation corresponds to the rule discussed in section 7. In particular the delay concept is illustrated within the task concept.

8.5 Usage Model, general simulation block and Architecture Model

The formulation of the complete simulation model includes modelling the architecture, the usage and describing the content of the general block. The usage is modelled with five processes describing the behaviour of the subscribers, both A- and B-subscribers, as well as a model of a call generator and a monitor which is responsible for creating the B-subscriber when a call is made, see figure 5 and the brief description above.

The processes in the general block are introduced to govern the simulation and to make measurements according to the objectives of the simulation. The measurements are specified so that they shall cover some usual measurement situations.

Finally, the architecture is modelled. This part is the most difficult in the example, since it includes a general data structure which handles the connections between the different models as well as the routing within the Architecture Model. The actual content of the structure is generated by one process and the process modelling the processor then works on the generated structure. The processor process is formulated so that it can handle the structure independently of the actual generation. The architecture processes are briefly described above. The data structure handling the connection between the models is based on linked lists. The structure is shown in figure 9 and it is briefly described below.

The structure consists of three queue types. The first queue (Proc_Queue) is a queue of the processors, i.e. the Architecture Model processes executing software processes (Software Model processes). This queue is the most central one. This queue is created by *Arch_Creator* and it is not altered during the execution of the simulation. For each processor two other queues are created, where the first one (Contact_i) models the connection of process instances from the Software Model and the Usage Model to the Architecture Model and the second one (Route_i) describes the routing. In the example this means five queues in total, i.e. one central queue with two objects (the two processors) and two queues for each of these objects. The routing queue of each processor is also static, while the queues containing process instances connected to the processors are dynamic, i.e. the contents of the queues are changed as new process instances are created or instances are terminated.

8.6 The simulation system

The modelling results in new SDL graphs describing the architecture, the usage and the general utilities. It is also necessary to alter some of the generated Software Model processes. These are changed due to the objective of measuring the load on the processors for each process type. The

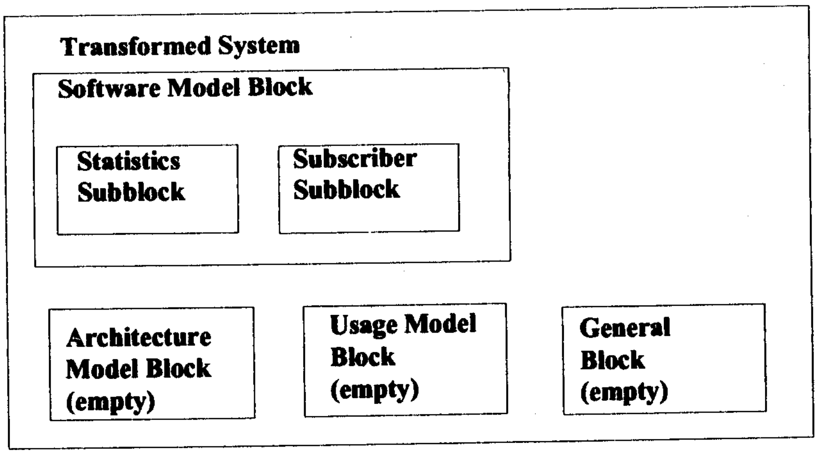

Figure 7: Transformed system.

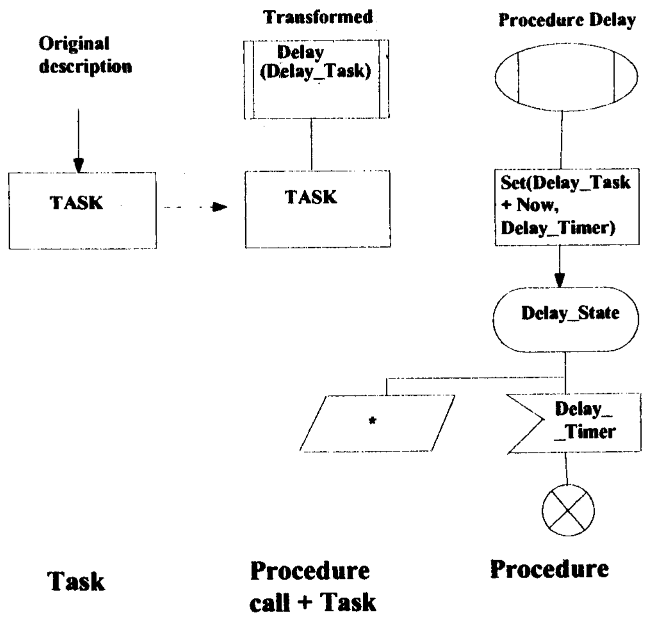

Figure 8: An example of a transformation.

additions are only minor, i.e. the graphs are complemented with a new variable describing the process type, which shall be used to measure the load on the processors for different process types.

8.7 Simulation results

The simulation system can be executed after having been analysed, generated, compiled and linked. It must be noted that the obtained simulation model cannot only be executed for performance and reliability analysis. It can also be used as a real time functional simulator. This means that the methodology provides a way of doing functional simulations in cases where it in the normal case is impossible (see above), i.e. for incomplete system descriptions (that is for example descriptions containing informal text in decision boxes). The transformation and generation facilitates execution of the original not completely specified SDL system from a functional perspective in a real time model environment as well.

The input data are not actual measurement data, but they have, however, been chosen to work as an input set where the relative size between the different inputs are reasonable. The main objective is as pointed out earlier not the actual values, but to show that the simulation actually can be performed based on the proposed concept. The values of the parameters are easily changed since they are declared as external synonyms in SDL.

Three parameters are of particular interest:

– The time, when the A-subscriber thinks he

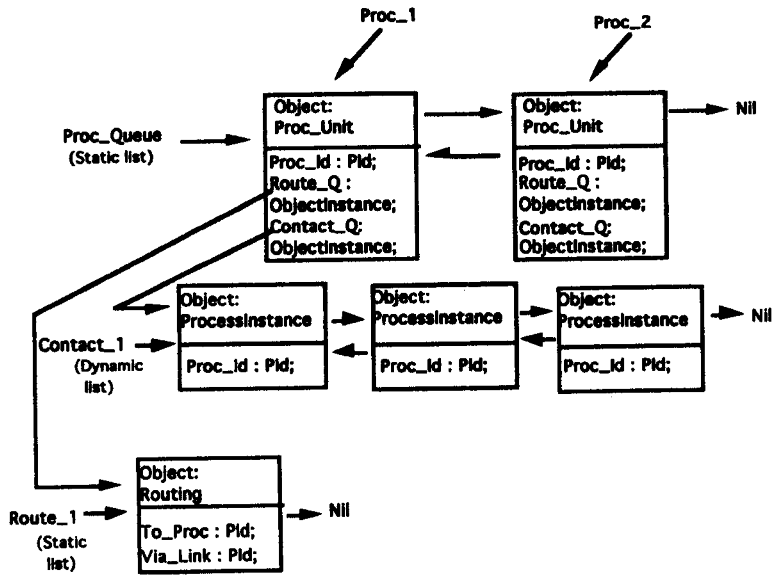

Figure 9: Data structure connecting the modelling concepts.

has waited too long before an answer, is first assigned the value 10, in which case the first output result below was obtained. The value is then changed to be equal to the simulation time.

- The simulation time is set to 1000. This may be too short to obtain real confidence in the output results, but since the actual figures are of minor interest it has been chosen short to obtain the results quickly.
- The mean time between arrival of calls is varied from 10 to 1.5.

The execution of the simulation model of the example gives three valuable results: 1) functional simulation results from a real time model, 2) reliability evaluation in terms of fault detection and 3) performance analysis results.

Functional result
A race between two signals is discovered, i.e. the behaviour of the system becomes different depending on the order of two signals. Due to the delay in the architecture it may happen that a terminate signal has not reached the receiving process instance before it sends a signal to the process that has terminated. This leads to a dynamic failure in the simulation. The original SDL system has been specified so that under some circumstances this will occur. Specifically the problem arises for high loads. A re-design is therefore necessary to cope with this problem, which would have been difficult to find without this simulation.

The functional fault would, without the method, probably not have been found until the load test in the test phase, since timing problems can not be evaluated with "normal" functional simulations. Since the usage profile is input to the simulation, faults will be found according to their probability of occurring in operation. Therefore the method will reveal faults as they are likely to occur in operation hence allowing for a reliability estimate in the same way as can be obtained through operational profile testing [11] and [12], and in statistical usage testing [5] and [6]. The simulation technique proposed advances though the estimation in the software life cycle to the software design phase.

Reliability result
The functional fault found is clearly a reliability problem at high loads. The fault found can hence be regarded as both a functional problem and as a reliability problem. Therefore it can be concluded that the software will at high loads be viewed as not being reliable, while in other cases it will be regarded as being reliable. No other faults were

found in the software during the simulation hence it was concluded that after correcting the functional fault found, the software can be approved, i.e. after regression simulation of the corrected software design.

Performance results
The results from the performance part of the simulation depend on what is specified by the user. The measurements are specified by the user of the system when complementing the generated simulation model with the Architecture Model, the Usage Model and the general processes that govern the simulation (see general block above). The latter has not been discussed in detail, but it is necessary to have some general processes for starting the simulation and perhaps for making measurement. These type of processes are quite simple to formulate and they will not be any problem for the user. In this particular case the load on the communication links was measured as well as the contribution to the total load on the processor by the different software processes. The latter was measured to identify the software processes contributing the most to the load and hence being candidates for re-design.

The results are shown in table 1. The table contains information about the mean time between new calls, the utilization of the two links, the total load on the most used processor, i.e. processor 1. The contribution to the load for the two processes that consume most execution power on processor 1 is also shown in table 1. An example for processor 1, when the mean arrival time between calls is 10; the figure 45 stands for the time process *A_Handler* executes and the time 176 is the total execution time used on the processor, i.e. out of 1000, which is the simulation time.

Some comments to the results in table 1 are worth making, even if the actual figures are of minor interest. It can be seen that the link from processor 2 to processor 1 is utilized more than the other, i.e. link 2. The reason is that the *Statistics* process is only located on processor 1. It can also be seen that process *A_Handler* is the largest contributor to the load. It is responsible for between 26-29% of the load, which means that a re-design of the process perhaps ought to be considered. The *Statistics* process contributes also very much and this is probably a problem, since the statistics in an exchange can be hard to motivate to the subscribers. A solution would be to distribute the statistics to all processors, and this will also cut down the utilization of the communication link between processor 2 and processor 1.

The obtained results show that valuable information relating to software quality can be obtained in the design phase with the proposed evaluation method.

9 Concluding remarks

Well-defined, formal or standardised description techniques provide excellent opportunities for automatic transformations to other representations. The other representation can either be a step in the development life cycle or a special representation for evaluating one or several quality attributes of the system. The presented method can be applied to both functional and object oriented description techniques. The quality attributes of the systems of today are becoming critical issues as the systems are getting larger, more complex and more critical. This means that techniques and methods for analysis of system quality attributes are needed to stay in control of the software system being developed.

This article has considered how software design descriptions and simulation models can be used to evaluate the performance and the reliability as well as the functional behaviour of the system at an early stage. In particular, the method provides an evaluation procedure before the coding and the testing phases.

The method provides a basis for:

- functional simulation in a real time model,
- estimation of software reliability in a simulation environment, when the usage in the simulation model is generated according to the operational profile,
- identifying software bottlenecks at an early stage,
- evaluating different distributions of software processes in an architecture,
- studying the introduction of new services in an existing system (network),
- examining different architectures ability to execute a given software description,

Mean arrival time calls	Utilization link 1	Utilization link 2	Load Pro. 1	Processor 1 A_Handler	Processor 1 Statistics
10	0.06	0.17	0.18	45(176)	51(176)
7	0.10	0.24	0.24	64(240)	61(240)
5	0.16	0.22	0.31	84(306)	72(306)
3	0.24	0.42	0.51	144(509)	106(509)
2	0.35	0.59	0.70	197(698)	140(698)
1.5	0.54	0.86	0.98	283(981)	181(981)

Table 1: Performance simulation results

– identifying system bottlenecks.

These issues will become important aspects as the demands on new services and systems grow in the same time as the requirements on short software development times and higher productivity continue to grow. The above list will be particularly important for safety-critical software systems, where a failure may be catastrophic. Part of the solution to meet the high requirements on functionality, performance and reliability is most certainly to put more emphasis on the early phases of the system life cycle through introduction of well-defined description techniques and methods that support different aspects of the development process. It is believed that methods for automatic translations of software descriptions into other representations will be one of the key issues to cope with the productivity and quality problems of software systems. The presented method provides an opportunity to tackle the problem of early verification of performance, reliability and functionality, as well as for doing capacity dimension

This method, or a similar one, is needed to control the quality attributes before the software system has been coded. The quality of the software must be evaluated and assured during early development. The presented method is a step in the right direction, towards a comprehensive view on quality control of software products.

Acknowledgement

Many thanks to Per Runeson, Q-Labs and Christian Nyberg, Dept. of Communication Systems, Lund, Sweden, for valuable help and constructive improvement suggestions concerning the article.

References

[1] R. L. Bagrodia and C-C. Shen, *MIDAS: Integrated Design and Simulation of Distributed Systems*. IEEE Transactions on Software Engineering, Vol. 17, No. 10, pp. 1042-1058, (1991).

[2] F. Belina F., D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specifications*, Prentice-Hall, UK, (1991).

[3] CCITT, *Recommendation Z.100: Specification and Description Language, SDL*, Blue book, Volume X.1, (1988).

[4] CCITT, *SDL Methodology Guidelines* Appendix I to Z.100, (1992).

[5] R. H. Cobb and H. D. Mills, *Engineering Software Under Statistical Quality Control*, IEEE Software, pp. 44-54, November, (1990).

[6] M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, (1992).

[7] E. Heck, D. Hogrefe and B. Müller-Clostermann, *Hierarchical Performance Evaluation Based on Formally Specified Communication Protocols*, IEEE Transactions on Communication, Vol. 40, No. 4, pp. 500-513, (1991).

[8] M. Håcansson and Ö. Persson, *Performance Simulation for SDT*, Master thesis, CODEN:LUTEDX(TETS-5176)/1-66/(1993) & Local 26, Dept. of Communication Systems, Lund Institute of Technology, Lund University, Lund, Sweden, (1993).

[9] B. D. Juhlin, *Implementing Operational Profiles to Measure System Reliability*, Proceedings 3rd International Symposium on Soft-

ware Reliability Engineering, pp. 286-295, Raleigh, North Carolina, USA, (1992).

[10] H. D. Mills, M. Dyer, and R. C. Linger, *Cleanroom Software Engineering*, IEEE Software, pp. 19-24, September, (1987).

[11] J. D. Musa, *Software Reliability Engineering: Determining the Operational Profile*, Technical Report AT & T Bell Laboratories, Murray Hill, NJ 07974, New Jersey, USA, (1992).

[12] J. D. Musa, *Operational Profiles in Software Reliability Engineering*, IEEE Software, pp. 14-32, March, (1993).

[13] P. Runeson and C. Wohlin, *Usage Modelling: The Basis for Statistical Quality Control*, Proceedings 10th Annual Software Reliability Symposium, pp. 77-84, Denver, Colorado, USA, (1992).

[14] M. Véran and D. Potier, *QNAP2: A Portable Environment for Queueing Systems Modelling*, Technical report, Bull, France and INRIA, France, (1984).

[15] W. Whitt, *The Queueing Network Analyzer*, The Bell System Technical Journal, pp. 2779-2815, November, (1983).

[16] J. A. Whittaker and J. H. Poore, *Statistical Testing for Cleanroom Software Engineering*, Proceedings 25th Annual Hawaii International Conference on System Sciences, pp. 428-436, Hawaii, USA, (1992).

[17] C. Wohlin, *Software Reliability and Performance Modelling for Telecommunication Systems*, Dept. of Communication Systems, Lund Institute of Technology, Lund University, Lund, Sweden, Ph.D Dissertation, (1991).

[18] C. Wohlin, *The Performance Prototyping Simulator Concept*, Technical report, IT4-project "Specification with Prototyping and Simulation", TeleLogic, Sweden (1991).

[19] C. Wohlin and P. Runeson, *A Method Proposal for Early Software Reliability Estimations*, Proceedings 3rd International Symposium on Software Reliability Engineering, pp. 156-163, Raleigh, North Carolina, USA, (1992).