A Software Maintenance Management Model based on Queueing Networks

Giorgos Papapanagiotakis
Peter Breuer

2487-TR-PRG-1062  Version 1.0
July 1991

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford  OX1 3QD
England

Copyright © 2013  P.T Breuer, G. Papapanagiotakis
Classification and Approval

Classification: Internal to ESPRIT REDO project no. 2487

This document is for use only with Project P2487 and may not be used for any other purpose unless specifically authorised in writing by the Project Manager, having obtained the agreement of the Project Management Committee.

Approved for issue as per above classification:

Author(s): Giorgos Papapanagiotakis
          Peter Breuer

Approved: P.T. Breuer

Date: July 1991

Doc. Id.: 2487-TR-PRG-1062 Version 1.0

Doc. type: Technical/Research Report


Work Package: 11

Task: 565

Distribution: REDO consortium, OUCL Library

Confidentiality: Freely distributable.

Document History
This is the freely distributable version of the REDO project document 2487-TR-PRG-1062.

Document Purpose
To describe the ISMMS Prototype built by G.P. at CTC Ltd. in Athens.

Acknowledgement
This report reflects work which is funded or partially funded by the Commission of the European Communities (CEC) under the ESPRIT II programme project no. 2487: "REDO: REengineering, validation and DOcumentation of systems"
Contents

1 The Software Maintenance Process: Conceptual Model 4
   1.1 Analysis Stage ................................................................. 5
   1.2 Processing Stage ............................................................. 6
   1.3 Testing Stage ................................................................. 6

2 The Software Maintenance Process Expressed as a Queueing Network 7
   2.1 Operation: How the Network Works ..................................... 7
   2.2 Environment: What Management has to Decide ...................... 9
   2.3 Information: What Needs to be Known ............................... 10

3 The Prolog Predicate Net Implementation of the ISMMS model 12
   3.1 The Prolog Implementation ................................................ 12
   3.2 Processing Tasks and Subtasks .......................................... 13
   3.3 System Support .............................................................. 14
   3.4 Database Implementation ................................................ 14

4 The ISMMS Prototype from the User’s Viewpoint 15
   4.1 Special Features .............................................................. 15
   4.2 The Software Maintenance Process: Running Simulations .......... 16
   4.3 Choosing Server Allocation Strategies ................................ 17
   4.4 Setting up the Database ................................................... 18

List of Figures

1 The ‘pinball’ model of software maintenance within the ISMMS. .... 2
2 The queueing architecture of the ISMMS processing model. .......... 3
3 An overview of the software maintenance process. ...................... 4
4 A more detailed flow-diagram of the software maintenance process. .. 5
5 Evolution of a queue over a interval of time ............................ 8
A Software Maintenance Management Model
based on Queueing Networks

Giorgos Papapanagiotakis* Peter Breuer†
July 1991

Abstract

A model of the software maintenance process within a medium-sized organization is described. We set out both the abstract model and its implementation as part of a management tool, the Intelligent Software Maintenance Management System (ISMMS) prototype.

The ISMMS provides a simulation environment for two sorts of management activities. In interactive mode, the characteristics of a particular set of maintenance tasks can be fed into the simulation, and decisions on the allocation of real personnel, tools and resources tested. In automatic mode, a spectrum of tasks is generated and different automatic allocation strategies are statistically evaluated against these. In both modes, the simulation draws on a database which is customized to accurately reflect the particular resources and characteristics of the organization using the model.

The abstract maintenance process underlying the prototype is described in terms of queues arranged in a topological network. This provides a clean design which has been successfully implemented using a three-layer software architecture in the prototype; the abstract queueing model is implemented as a synchronized predicate-net which in turn is implemented in Prolog, and we describe the implementation here, as well as its behaviour from the user's point of view.

Introduction

Software maintenance is a developing field which is being actively researched [2], [9], and this makes it a difficult area for managers to manage quantitatively; there is not yet a ‘classical’ science to draw upon which can provide the theoretical models and the numbers required. But no-one can wait for a revelatory theory to arrive - the software explosion is with us, and it brings with it its own maintenance mountain. It is urgently necessary to manage the expertise of organizations in such a way that they will make best use of the resources and tools available now, whether or not these will, with hindsight, eventually become recognized as the ‘correct’ tools for the job.

Accordingly, we have adapted the general techniques of process management control [1] to the maintenance process as we believe it is to be found within a software house. We have produced a general model, and a software implementation as a working prototype tool, the Intelligent Software Maintenance Management System (ISMMS), which may be tuned across a broad parameter range. The ISMMS can provide information on the consequences of

- assigning particular individuals to particular kinds of tasks (in interactive mode), or
- adopting certain general management strategies (in automatic mode).

*Pandrive S.A., 19 P. Ralli Avenue, 17-77-8 Athens, Greece, and formerly at CTC Ltd, Athens, Greece, where this research has been funded under ESPRIT II project REDO, No. 2487, on the Maintenance, Validation and Redocumentation of software systems.
†Oxford University Computer Laboratory, 8-11 Keble Road, Oxford OX1 3QD, UK, also funded under project REDO.
It is a crucial feature of the ISMMS that it makes use of databases which reflect in detail the personnel, tools and resources of a particular organization – this is what was meant by ‘parameter’ above – and a crucial feature of the abstract model that it is generic in such values.

The underlying concept of maintenance management found here is based on the ‘good practice model’ introduced by K. Bennett [3], [7]. According to that, software maintenance is considered to be the modification of a software product after its final delivery in order to either

i) correct faults, or

ii) improve performance or other attributes, or

iii) meet new requirements (i.e. adapt the product to a changed environment),

and the maintenance model presented here deals with the allocation of resources for just this kind of maintenance request. It is not a conceptual model, but rather is a mathematical and computational model of an extremely general organizational pattern which we believe fits the maintenance departments of most software houses.

In general, we may say that we require information of two sorts:

• that which relates to the software maintenance department itself

• that which relates to certain measures of the software to be maintained

This information is stored locally in three databases within the ISMMS, as detailed in section 3.2. Briefly: database

A) contains information on the characteristics of service agents,

B) contains information on the tools available as resources to the agents,

C) contains information on the characteristics of incoming maintenance tasks.

These databases are customizable. The model is made particular to a given department by fixing the number, type and quality of ‘service agents’ available, and fixing the number, type and quality of ‘resources’ available to them.

![Diagram of software maintenance model](image)

Figure 1: The ‘pinball’ model of software maintenance within the ISMMS.

In a software maintenance department, the ‘service agents’ are people (or aspects of people), and the ‘resources’ are software toolsets and workstations, but for the purposes of the abstract model, they can be treated as processes and resources in the JSD [6] or CSP [5] sense, which is to say; as ‘machinery’ with a certain throughput which depends on the kind of input they receive, as well as the resources they use to accomplish it all, and the intrinsic characteristics of the ‘machinery’ itself. The abstract problem which
we treat in the ISMMS design is how best to distribute incoming maintenance tasks to available service agents against a particular background of resources, and putting things into this abstract setting allows us to apply quantitative modeling techniques to the problem.

Now, at the gross level, a software maintenance department is a system which receives maintenance requests, processes them, and returns the completed tasks (figure 1). In section 2 we will describe the detailed general process model, but, generally we can say that the ISMMS model uses the intrinsic characteristics of an incoming maintenance task in order to divide it up into subtasks. These subtasks fall into categories at which particular ‘service agents’ may individually be better or worse than others, so performance is dependent on who works on what, and on how – which tools are used for the job. Moreover, the maintenance process is modeled in the ISMMS as taking place in several distinct stages, divided into phases, and the model requires that a ‘service agent’ be responsible for a unique subtask at each processing phase as it progresses through the system, although a different agent may be responsible for a subtask at different phases. On this model, the frequency of ‘binding a new service agent to a subtask’ is necessarily high – subtasks will appear to be ‘bounced around’ within an active and efficient system at high speed – and therefore the allocation mechanism within the ISMMS is extremely important (each such mechanism corresponds to a possible strategy within the maintenance department). In section 3, when we describe the design of the ISMMS we will pay particular attention to the incorporation of these allocation mechanisms. The built-in strategies and options are sketched in section 4.

The abstract model of the software maintenance process takes the form of a queueing network with workspaces, plus a server pool. The whole system is essentially a network of queues (infinite buffers) connected together in a particular way, plus an unstructured set of (available) service agents, working as follows (see figure 2): Each phase of the maintenance process has an associated queue, and subtasks pass from one queue to another as they pass through the system. A subtask queued for treatment in a particular phase of the maintenance process will eventually be allocated a free service agent from the pool, at which point it leaves the queue and is treated by the agent in a workspace ‘adjacent’ to the queue. After a certain period of logical time, the treated subtask is released by the agent and it proceeds from the workspace to the next queue in line, whilst the agent migrates back to the pool, where it is again available for work.

Task Queueing Network

Only at one point in the network is the simple queue-plus-workspace structure deviated from, and this is at the assembly point of the maintenance process, where tasks must be reassembled from their component subtasks. Here we use a slightly modified queue structure in order to represent the synchronizing functionality correctly and efficiently. But otherwise, the network connects simple queues to simple queues, and that is all.

The queueing network model may be specified within the formalism of predicate nets, a general
transition-based formalism able to express priorities, synchronization, variable queue lengths and temporal relationships, and this is the route to an implementation which we have followed. The predicate-net formalism has been interpreted in Prolog, as have the remainder of the working parts of the prototype software maintenance management tool: the predicate net queueing network model for the maintenance process, the databases of departmental resources and application software measures, and the allocation strategies.

The plan of this article is as follows: In Section 1 the conceptual model underlying the maintenance model is described. Section 2 deals with the expression of the model in terms of a queueing network. Section 3 describes the implementation of the queueing network as a predicate net, including the representation of this net in Prolog. Finally, Section 4 deals with the user’s view of the ISMMS prototype.

1 The Software Maintenance Process: Conceptual Model

In this section we describe the conceptual model behind the ISMMS in a top-down fashion.

Maintenance requests are generated externally by application users and input to a software maintenance department as a single maintenance task. The term ‘maintenance task’ refers to a single piece of software together with a specification for changes to be made to it, but, since both the software and what remains to be done to it change as maintenance takes place, it is best to think of a task merely as a convenient and constant identifier.

Tasks pass through a maintenance process which has several stages, each of which may be classified under three main headings; analysis, processing and testing:

Analysis. The maintenance task is scrutinized and initially accepted or rejected. The reasons for this decision may be purely technical, or they may represent current departmental policy, but this decision-point acts as an early filter. If accepted, the cost of the work to be carried out and its cost and priority are estimated, and then a detailed breakdown of the task into maintenance subtasks is prepared and a maintenance plan is evolved for each. The term ‘maintenance subtask’ refers to a software module (possibly a high-level module, which manipulates complete programs or modules itself, or possibly just a single low-level subroutine) together with a specification for changes to be made to it. Again, it is best to think of a subtask as a constant identifier which is bound to evolving software and change-specifications in order to track it and provide a handle to refer to it with.

Processing. The subtasks are passed to different sections of the department and the planned maintenance is carried out, possibly in parallel.

Testing. Testing starts with tests of the changes carried out in each maintenance subtask, and continues after integration to test the complete module and the final release.

The three stages of maintenance take place approximately (or at least ‘conceptually’) in sequence, and each stage is in turn divided into phases which are intended to capture the natural process-flow as the maintenance task passes through the technical services department.

The maintenance process may as a first approximation be described as a linear flow-diagram, as in figure 3.

![Figure 3: An overview of the software maintenance process.]

But a more detailed picture of the process will show a degree of parallelism. The first stage (analysis) indeed consists of phases which lead into one another sequentially. But, finally, a ‘multi-output’ phase
in the analysis stage is reached, where the task is split into subtasks which are simultaneously placed on several queues at the head of the *processing* stage of maintenance. The complete flow-diagram has a fixed topology, and it is shown in figure 4.

![Flow-diagram of software maintenance process](image)

Figure 4: A more detailed flow-diagram of the software maintenance process.

The three strands to the *processing* stage shown in figure 4. are purely for the purpose of illustration – each represents a processing route for a known class of subtask. The model is not restricted to three (and in fact the ISMMS prototype implementation defaults to four). Nor, for reasons of clarity, have quite all the connections been drawn in in the figure: there are also some by-pass and feedback loops. There is an additional by-pass from the accept/reject phase to the output and feedback loops from the test phases to appropriate prior phases in case the processed software fails the tests. The diagram is accurate apart from these minor omissions, and faithfully represents the straight-through path of any maintenance task which is completed smoothly.

The detail of the various stages and phases of maintenance is expanded upon below.

### 1.1 Analysis Stage

This consists of:

**Accept/reject and refine-specification phase**

A maintenance request is examined from the technical point of view and is either accepted or rejected as technically inconsistent. A management decision may also influence this phase – authority for further processing may be denied for reasons of policy beyond the technical aspects alone. If the maintenance request is accepted, a better and more detailed definition of the problem will probably be needed from the originator of the request.

The service agent during this phase is a maintenance engineer with some managerial responsibilities.

**Cost estimation and prioritizing phase**

Software complexity metrics are applied to the application software, and the results are entered in the SW application database in order to calculate a *maintainability index*. The difficulty of the maintenance request is then estimated and the cost (human effort, use of SW/HW environment, etc.) is calculated.

A *priority* is assigned to each request according to criteria such as urgency, genericity, and the price the customer will pay. This price offsets the cost of the service for the maintenance department, but we do not suppose that the sum is required to be positive. Of course, in practice the price is negotiable in that a higher priority may be assigned if a higher price is acceptable. The final estimate of net cost may modify management decisions and managerial approval for the continuation of request processing may be necessary at this point.

The service agent for this phase is a maintenance engineer with a great deal of managerial responsibility.
Decomposition and planning phase

The request is analyzed by technical experts to identify which parts of the application software will be affected. A detailed decomposition of the maintenance request into several subtasks is carried out, each subtask to lie entirely within the ambit of one of the classifications for maintenance processing. For example, most probably several subtasks which are associated with low-level code changes will be generated, and somewhat fewer subtasks which have to do with linkage-control and inter-module communication will be generated. These two types would normally be classified as lying in two different ‘layers’ and would be treated differently (i.e. by different people and methods) within the department.

The service agents are experienced maintenance engineers with wide knowledge of the entire SW application and maintenance background. The most experienced engineers analyze the most difficult maintenance requests.

The individual subtasks are then simultaneously queued at the inputs to the processing stage under the classification (A, B, C, etc. in the ISMMS prototype) appropriate to the subtask.

1.2 Processing Stage

Maintenance tasks are processed in parallel under the classification which they have been given, in the order in which they arrive in the queue for their class.

The service agents are maintenance engineers using resources from the SW/HW environment. Particular engineers and/or resources may be better suited to service particular classes of subtask.

1.3 Testing Stage

The ‘final’ stages of software maintenance is devoted to testing. Modules are tested (simultaneously, in parallel), then assembled and tested again. The phases of this stage are as follows:

Module test phase

Software modifications are tested locally after they have been implemented.

The service agents are maintenance engineers with experience in the class of maintenance which has been carried out, and, more generally, in software testing and the use of the appropriate testing tools in the SW/HW environment.

At this point, subtasks are re-assembled into complete tasks again.

Integrate test phase

Communication and data-sharing, and other aspects of control (such as synchronization) between those modules affected by the modifications is tested.

The service agents are maintenance engineers with experience in SW integration testing and possibly also experienced with the SW modules under integration test and with any required SW/HW environment tools.

Release test phase

Application software is tested as a whole in order to check that the requirements are satisfactorily met.

The service agents are software engineers with experience in the particular SW application and also both the maintenance and target SW/HW environments.

In the next section, we go on to express the conceptual model whose description we have just completed in terms of a queueing network. This is the intermediate layer in the three-layered model which this article describes.
2 The Software Maintenance Process Expressed as a Queueing Network

The intermediate layer of the software maintenance process model is organised into three major components:

i] a queueing network, with

ii] workspaces, and

iii] a server pool.

That is, a triple: (queue, workspace, pool). But this is not all – the software maintenance process interacts with its environment, in that decisions are sometimes required from managers, and ultimately maintenance requests are both received and released into the environment. And the process works against a background of information which characterizes the process in one software maintenance department in comparison to the process in another.

In the following sections we deal respectively with the operation (the design of the network model), the environment (principally the decisions required of managers by the model) and the information (the database which supplies all the relevant parameters) which make up the complete design.

2.1 Operation: How the Network Works

The queueing network with workspaces is a network of what are essentially infinite buffers (queues) connected together in a fixed way. But each queue is associated with a devoted portion of the workspace, so the topology of the queueing network imposes a structure upon the workspace too. The server pool is an unstructured set of (available) service agents. The model implicitly incorporates a highly parallel semantics, which we sketch below:

Each phase of the maintenance process described in section 2 has an associated queue, and subtasks pass from one queue to another as they go through the process. A subtask queued for treatment in a particular phase will eventually be allocated a free service agent from the pool, at which point it leaves the queue and is treated by the agent in the workspace ‘adjacent’ to the queue (see diagram 2). After a certain period of logical time, the treated subtask is released by the agent and it proceeds from the workspace to the next queue in line, whilst the agent migrates back to the pool, where it is again available for work.

All this activity takes place simultaneously. During a particular unit of logical time, many of the following transitions can take place, but no others:

• a service agent which is engaged in servicing a task or subtask may continue with the service for one unit of time, at the end of which it may either release the task onto a queue, and return to the pool, or continue servicing.

• a free service agent from the pool may leave the pool and bind to a task or subtask waiting at the head of some queue, remove it from the queue, and process it for one unit of time, at which point it may either release it onto a queue again, and return to the pool itself, or it may continue processing.

It is to be imagined that the binding of a server to a task or subtask takes place at the beginning of a unit interval of logical time, and the release occurs at the end of an interval. As a consequence, a server cannot perform a service which lasts less than one unit of time.

However, a server which releases a task becomes available immediately for another service. Or, putting it from the other point of view, a task which has been serviced in a particular phase of maintenance may immediately proceed to be serviced in a next phase.

A consequence of this model for the behaviour of the queues is that, since all the removals from queues occur at the beginning of an interval, and all the accessions occur at the end, over the interval the queue evolves as shown in figure 5: a tail segment becomes an initial segment.

We have stated that the queueing network consists essentially of queues. All of the data-structures associated with the nodes of the network are simple queues, except one: at one point only is the simple
queue-plus-workspace structure deviated from, and this is at the assembly point (see figure 4.), where tasks must be reassembled from their component subtasks. Here we use a slightly modified queue structure to represent the synchronizing functionality correctly. But otherwise, the network connects simple queues to simple queues. The overall design has been illustrated in figure 4, and the diagram imparts a precise semantics to the model, as follows:

- Where a single queue is connected by the network to many successors, the service-agent may release subtasks to one or more of these.
- Where one queue is connected to a unique successor queue, then accession is inevitable provided that the release criteria specified for the service is satisfied, otherwise accession is blocked, and the subtask cannot leave the workarea and its service agent remains bound to it – the intention is to allow only subtasks which are completed ‘satisfactorily’ to proceed.

After the end of the processing stage, at the beginning of the testing stage, there is a node in the network which has several inputs. Here, subtasks are reassembled into tasks again, before proceeding through to a sequential series of tests. The ‘assembly point’ node is the convergent of a many-one linkage, and we interpret it in a slightly special way.

Notionally, subtasks on the incoming links are bound to service agents which process them and then must pass them on. However, two schemes for the ‘end-service transition’ are both (computationally) possible and reasonable:

i] they can immediately release the subtask onto the assembly queue, or

ii] they can wait for all the other subtasks of the same task to become ready for assembly, and then co-operate in releasing the component subtasks simultaneously onto the assembly queue.

Because we do not wish to bar agents from releasing subtasks as soon as possible (we wish to allow the agents to become free for other work) so we do not follow scheme [ii]. However, [i] requires that subtasks be assembled (‘synchronized’) in some other way, and here again we have two options:

- We can give the assembly point (and, more generally, nodes which are the convergents of several linkages) a more complex structure than a ‘simple queue’, which allows them to act as synchronization points for a group of associated jobs without tying up service agents in the synchronization process.
- We can stick with the simple queue structure, and assign the ‘secretarial work’ of assembly to a large team of service-agents especially charged with this sort of work, and whose time we do not mind wasting.

We choose the first option, and not the second. The structure assigned to nodes at the convergence of many-one linkages is a ‘queue-plus-accumulator’. Subtasks pass into the accumulator until all the components of the complete task are available there, whereupon it proceeds automatically as a whole into the queue part. There it becomes available for further processing by another service agent as normal.
2.2 Environment: What Management has to Decide

Recall that the queueing network model adopted here envisages an incoming maintenance request as

1. being divided into many separate maintenance subtasks, which are

2. subsequently allocated, synchronized amongst the different phases of the software maintenance processing model, ultimately

3. being reassembled and passed out as completed.

The model depends crucially on the notion of an ‘indivisible subtask’. A maintenance subtask is a piece of work carried out during a maintenance phase which requires the allocation of specific personnel and SW/HW utilities. It is a fundamental unit – it cannot be of such a size as to require ‘half a utility’, or ‘half a server’ – and the users interaction with the ISMMS model (in interactive mode) always concerns the allocation of resources to a subtask, and to no other unit. Occasionally, the ISSMS model requires decisions to be made and returned to it on alternatives that it has offered up for choice.

In general, there are two types of management decision which may have to be made:

i] personnel decisions, concerning the allocation of suitable personnel to the subtasks of the maintenance task.

ii] SW/HW decisions. concerning the allocation of a set of SW/HW environment utilities.

Usually (that is, in the real world), personnel allocation will take place first. That is the situation in the ISSMS model too.

In order to take decisions of this kind, managers using the ISMMS will consult the information stored in the three databases:

A Personnel Data Base: One looks here to check on the characteristics of the available service agents in order to choose one of those offered by the ISMMS for a given subtask.

B Toolset Data Base: One looks here to check on the kind of resources needed for a given subtask, bearing in mind that allocation of scarce resources may hinder processing of other subtasks.

C Application SW Data Base: One looks here for general and specific information which is useful for evaluating the complexity of the maintenance task

The procedure a manager should follow – and the procedure which is emulated by the ISMMS in automatic mode – is as follows. Whenever the ISMMS model offers up a given subtask for servicing, the available personnel ought to be placed into groups \( g_1, g_2, g_3, \ldots \) of equally effective personnel by the manager, according to their suitability for that particular subtask. This is the easiest way to start making decisions about allocations – no ordering is required, just ‘is as good as’ opinions. In a second phase, one may order the groups of equivalent personnel by preference:

\[ g_1 \geq g_2 \geq g_3 \geq \ldots \]

In fact, these preferences can be decided ahead of time, one grouping and ordering for each class of subtask.

SW/HW utilities should also be grouped into sets \( ts_1, ts_2, \ldots, ts_m \) by the manager. This time, the idea is that each set consists of all those utilities which will be needed to perform a specific class of maintenance subtask. In composing these groups, managers should avoid allocating the same tool to different toolsets to be certain of interference-free working, but partitioning may cause operation of the model to be slowed. This classification can once again be carried out ‘offline’, simply on the basis of the attributes of those subtasks which may occur. In practice, the best result that can be hoped for is an assignment which achieves the reasonably free passage of most subtasks through the processing system.

The net result of this systematic classification, carried out by a manager, is at least to inform which personnel and which resources are suitable to allocate to a given maintenance subtask. But the particular decision-making mechanisms employed by a manager in making a final choice, given this information, may never be absolutely specified. In the ISMMS, they are either left to the manager (in interactive mode), or
specified (in automatic mode) but different strategies may be added by the user, or different pre-supplied strategies may be activated.

For example, the naive strategy allocates to each maintenance subtask the most suitable (best ranked) personnel and toolset currently available. This strategy is one of the built-in, pre-supplied strategies in the ISMMS. But this is a non-predictive strategy. A different kind of strategy is to optimize a particular measure. For example, if the following items are costed:

- labour
- SW/HW tool use
- delivery time (lead/lag)

a viable management strategy would be to allocate personnel and tools so as to minimize a weighted sum of the costs involved in the maintenance request (including a component based on the total predicted processing time given the proposed allocation).

Several such strategies are built into the ISMMS, and will be detailed later. Below, we describe the nature of the information on which these strategies are based.

2.3 Information: What Needs to be Known

At different phases of maintenance processing, the servers for a task or subtask may be either software engineers or tools from the SW/HW environment. In order to make the selection and scheduling of resources possible, there is one database in the ISMMS for each of the above kinds of server, plus one database which holds information on the classes of application codes, obtained from qualitative and quantitative evaluations, perhaps using metrization tools.

Recall that these three databases are

A  a software maintenance personnel database
B  a SW/HW environments and toolsets database
C  a database holding information on the characteristics of different types of application software.

There follows a more detailed description:

Maintenance Personnel

This database contains personnel profiles which permit the informed selection of the appropriate maintenance engineer for a given maintenance task. Further, current information including the nature and priority of any work that may be already under way is included here.

The database lists various attributes, each of which may be classified under one of four general categories, as follows:

- General qualifications
  - Experience in maintenance
  - General programming experience
  - Experience in programming in the source language of the application

- Management- and organization-specific criteria
  - Experience in cost estimation for maintenance requests
  - Experience in prioritizing requests
  - Experience within the organization
  - Experience working on maintenance for this organization

- Software application-specific criteria
– Skill in the technical analysis of maintenance requests
– Skill in maintenance directed at organizational level \( n \) (system, module, function, etc.) of software applications.
– Skill at testing directed at organizational level \( n \) of software applications.
– Skill in integration testing
– Skill in release testing

• SW/HW environment- and toolset-specific criteria
– Experience in using toolset \( n \)

The database must be created and maintained by the managers. There will be rather infrequent modifications and we consider the database to be constant in the ISMMS model.

The list of database attributes above is surely not exhaustive, and may need to be modified for different organizations’ particular needs. For example, one may want to include information on holiday periods, and so on.

One will wish to define procedures which interrogate this database and output a list of maintainers that best fit the particular maintenance task. The ISMMS prototype implementation has ‘first fit’ procedurality built into it, but a more intelligent algorithm may be substituted. The algorithm used will almost always be sub-optimal in practice.

**SW/HW Environments and Toolsets**

This database contains information which will aid the selection of a particular maintenance utility as a server. It also contains information on the current allocation status of the utilities.

The stored attributes are:

• Type of utility (metrization tool, structure editor, etc.)
• Performance characteristics (fast, slow, good, bad, etc.)
• SW/HW requirements (dedicated workstation, proprietary database, etc.)
• Utility usage costs!
• Current and planned availability.

An algorithm which identifies the best suited set of utilities for the performance of a given maintenance task is required in the ISMMS model. The prototype has the ‘first fit’ procedurality built into it, but another may be supplied.

**Application Software**

This database contains information on particular and general classes of application software. This is essentially a

• Classification (editor, spreadsheet, operating system, payroll, etc.)
• Complexity index (complex, easy, etc.)

This information is used to estimate the complexity of incoming maintenance tasks and establish a threshold for skills and abilities of the servers which may set to work on the task.
3 The Prolog Predicate Net Implementation of the ISMMS model

Predicate Nets (PN) make up a transition-oriented formalism based on Petri Nets but allowing a greater range of datatypes to be handled, and greater control over the triggering conditions for individual transitions.

Each transition of a Predicate Net is specified by

- **Pre**: a predicate which acts as a firing condition (pre-condition), and
- **Post**: an action, phrased as predicate expressing a post-condition, and
- **Data**: a set of predicates expressing data-conditions.

The Predicate Net formalism is used within the architecture of the ISMMS to define the details of the queueing network model. For example, the assignment

\[ x := x + 1 \]

might be represented as the triple

- **Pre**: \( x = x \)
- **Post**: \( x = x + 1 \)
- **Data**: \( x \) has been (declared and) initialized

Queue structures are easily expressed in PN terms [8], [4]. A queue may be thought of as a list data structure with certain access operations, and scheduling and synchronization may be expressed using data conditions.

3.1 The Prolog Implementation

In the ISMMS prototype, a Predicate Net has been implemented in Prolog to provide the appropriate functionality. The queues are represented as predicates

\[ q_X(x_1, x_2, \ldots) \]

asserting that the \( x_i \) are the elements on the queue. The tasks (and subtasks) which pass through the network are implemented as predicates

\[ t_Y(\text{name}, \text{priority}, \text{value}) \]

which assert their name and priority. The predicates necessary as preconditions, postconditions and data conditions are written as Prolog clauses.

The overall operational semantics of the implementation may be described through a coarse-grained Petri Net as follows:

- **Pre**: Maintenance requests, which may be thought of as having just arrived from the external world, are cached in the queue \( q_{\text{In}} \). The processing of the \( n \)'th request is initiated in the model at the time \( t_n \) asserted by the predicate

\[ \text{Generate}(t_1, t_2, t_3, \ldots) \]

- **Post**: The completed maintenance tasks are placed on the queue \( q_{\text{Out}} \). The rejected requests (rejected during the Accept/Reject phase) are placed on the queue \( q_{\text{Rej}} \).

- **Data**: Service activities (services) must have been triggered in sequence on tasks and subtasks, which have been transferred from queue to queue through the maintenance process.
3.2 Processing Tasks and Subtasks

The order of phases in the processing of all maintenance tasks (and subtasks) is expressed through Predicate Net transitions of one of the two formats

transition Start_Service.X

Pre : \ldots q_A \ldots
Post: \ldots q_A \ldots

or

transition End_Service.X

Pre : \ldots q_B \ldots
Post: \ldots q_B \ldots

where A and B are the names of phases with associate queues, and X stands for the name of the service that is required to move a task from A to B.

The preconditions of all Start_Service transitions capture the current system time, ascertain which server is free, and look at the head of the associated queue. The postconditions assert (force) the system time to be updated – i.e prevent the service being completed before the given time – and assert that the intended service X has been performed by the server.

transition Start_Service.X

Pre : \text{Clock}(\ast\text{start}\_\text{time})
Server(\ast\text{server})
q_A(\ast\text{task}, \ast\text{more})

Post: \text{Clock}(\ast\text{start}\_\text{time})
Service(\ast\text{task}, \ast\text{server}, X, \ast\text{end}\_\text{time})
q_A(\ast\text{more})

Data: duration(X, \ast\text{server}, \ast\text{service}\_\text{time})
skill(\ast\text{server}, X)
\ast\text{end}\_\text{time} = \ast\text{start}\_\text{time} + \ast\text{service}\_\text{time}

The data conditions check that the server has sufficient skill to carry out the service X, and block the release of the task by the server whilst the current time is still not the calculated release time for the bound pair.

The net effect of the successful firing of a Start_Service transition is always to do the following

1. pop the appropriate queue
2. bind a free server
3. capture the current time according to the clock
4. transmit the task for further servicing
5. calculate the time when servicing will be completed (a function of the service requested and of the allocated server).

Likewise, the preconditions and postconditions of End_Service transitions specify and proscribe the state of precisely the same entities, but this time it is the precondition which states that the service has been completed in the correct amount of time, and the postcondition which states that the server is free – because it has now been unbound from its task or subtask.

transition End_Service.X
Pre: Clock(*end_time)
  Service(*task, *server, X, *end_time)
  \(q_B(*queue)\)

Post: Clock(*end_time)
  Server(*server)
  \(q_B(*newqueue)\)

Data: priority(*task, *priority_value)
  schedule(*task, *priority_value, *queue, *newqueue)

The data conditions here simply pull the priority of the task out from its structure (each task is represented
as a predicate asserting a certain priority together with its processing label) and ensure that it is replaced
onto the queue in the correct position relative to all others on the queue.

So the firing of an End_Service transition always has the following effects:

1. release the task or subtask from service at the appropriate time.
2. frees the appropriate server
3. transmit the task or subtask to the next queue.

This completes the specification of the topology of the network, and leaves only the definition of individual
services to be filled in.

3.3 System Support

Time is expressed through a predicate:

Clock(*time)

where *time indicates a logical time. The predicate will assert a unique current time for the system.

Time evolves independently of anything else, so the transition in the Predicate Net model which
represents ticks of the clock must have the lowest priority, in order to ensure that everything which could
be done in a given period is actually attempted before the clock moves on!

The various maintenance subtasks of a maintenance task leaving initial testing must be synchronized
before they proceed on to the ‘Integrate Test Phase’ in the model. An extra queue and the Petri Net transition

SYNC

are used to specify this synchronization mechanism, and together they simulate an ‘assembly area’ in the
neighbourhood of the assembly-point node in the process diagram.

3.4 Database Implementation

The ISMMS research prototype contains small databases implementing part of the full range of knowledge
envisaged in the general model. They store information about (A) personnel skills, (B) the available
toolsets, and (C) about incoming application software. These databases are relational schemas composed
of only one relation each.

The personnel database is comprised of the relation:


keyed on *server and *phase. It delivers a rating for each server with each toolset classification in each
phase of the maintenance process.

The toolset database has the following relation:

toolgroup(*tool, *phase, *application_type, *tool_group)
keyed on the first three columns. It delivers a classification for the tool in particular circumstances.

The application software database contains the two relations:

- **difficulty(application, phase, tool_group, difficulty_rating)**
- **applicationtype(application, application_type)**

keyed on the first three and the first columns. The **difficulty** relation delivers an intrinsic difficulty of maintenance per phase, using a given classification group of tools, and the **applicationtype** relation delivers a general classification for the application.

The above relations are implemented as functionally deterministic predicates in the prototype.

4 The ISMMS Prototype from the User’s Viewpoint

The Intelligent Software Maintenance Management System (ISMMS) has been implemented as a research prototype, set out here. The ISMMS prototype is intended to assist the maintenance department manager of an organization to properly allocate the available human resources of his department to incoming maintenance tasks, either through simulation (in **automatic** mode) or through interaction (in **interactive** mode).

4.1 Special Features

The queueing network model underlying the model introduces some particular features in comparison with the more queueing network models frequently encountered in data communication, telephony and operational research:

- **Floating Servers**: In the ISMMS model there are several potential servers who may service a particular maintenance task or subtask. In contrast, OR networks usually associate a single server to a single queue uniquely, because they seek to model a layout akin to a factory floor, in which tasks pass from fixed station to fixed station, and there is normally an index of the servers ability to service the queue (usually this is the mean service time) which will determine the behaviour of the pair in response to any given distribution of requests.

- **Customizable Binding Strategies**: Therefore managerial decisions are required to dynamically bind an available server to a maintenance task at a given site (phase of processing). There is an index of suitability for every pairing and the decision can produce either a binding with a low index or a binding with a high index. In contrast, the stations in OR networks usually have a fixed performance index, because they represent machinery with a fixed operating cost or efficiency.

  There are several management strategies which could be used to make the decisions required and some are built into the ISMMS prototype (detailed below). These strategies can vary from the random to the sophisticated, possibly optimizing some organization-oriented criteria. In general, such criteria may include: maintenance time minimization, cost minimization, fair load distribution amongst servers, etc.

- **Synchronization**: The ISMMS model has built into it a requirement for the synchronization of the disassembled subtasks before they are reassembled again for integration testing. This feature can be present in OR models (for obvious reasons!).

- **Prioritization**: Every maintenance task on a queue has a separate priority, and tasks and subtasks queued for processing are offered up for service in order of priority. The priority is assigned to a task as a management decision early on.

- **Low Throughput**: There are only a relatively few service requests which will be entered into the model per ‘day’. This reflects the expected frequency of maintenance requests in a small- to medium-sized software maintenance department.
From the user's point of view, the ISMMS prototype may be thought of as a particular software maintenance department, and it may be divided conceptually into three parts:

- the software maintenance process itself, as simulated by the model.
- the server allocation strategy used to assign servers to subtasks and services.
- the stored knowledge about available servers and the incoming application softwares.

We detail how the user interacts with or sets up with each of these three below:

### 4.2 The Software Maintenance Process: Running Simulations

As sketched above, the department's maintenance request process is implemented in the prototype as a queueing network, defined in terms of a Predicate Net which runs in logical time, servicing tasks queued at each of the following phases of maintenance at each tick of the logical clock.

1. Accept/reject maintenance request.
2. Cost estimation and priority allocation.
3. Analysis into subtasks.
4. Carry out maintenance at level \( n \).
5. Test at level \( n \).
6. Integration test.

Now, the user can operate the ISMMS prototype in one of two states: *Interactive* or *Automatic* mode. The default is *Interactive*.

In this mode, the user is required to take those decisions interactively which would otherwise be made for him automatically when the simulation runs in *automatic* mode. The decisions which are required will of course be made only after consulting with the database and looking at the display windows, but they are the following:

- An ‘accept or reject’ decision is required at the initial accept/reject phase of processing.
- A priority assignment is required at the (cost estimation and) prioritization phase. The priority is an integer between 1 (lowest priority) and 3 (highest priority).
- The level at which maintenance must be carried out has to be identified at the analysis phase. The ISMMS prototype permits up to four levels – A, B, C and D – to be designated for each incoming request. The ‘levels’ are intended to denote the range between ‘lines of code’ and ‘system organization’.
- An ‘accept or reject’ decision is required at every test phase.

Moreover, in the interactive mode, the user is required at each tick of logical time at which a subtask is queued for processing and a valid server is free to begin servicing it, to actually make the decision to bind that server to that subtask (at that phase). This may entail a choice from amongst several possible servers, or several possible subtasks (waiting at the head of several different queues). The ISMMS prototype automatically restricts the possible choices to those servers who are suited to both the service and the subtask, according to the database. This may mean that a particular server may not be offered as a candidate for a particular service even though it is free.

The ISMMS prototype automatically frees an allocated server at the conclusion of the processing subtask on which it is engaged, after a logical elapsed *service time* which is determined by the intrinsic difficulty of the maintenance phase and the servers abilities in this phase.
The user can always reverse a series of decisions, returning the models state stepwise to an earlier time. In the automatic mode of operation, accept/reject decisions, prioritization and server allocation is performed according to strategies selected beforehand by the user. A standard set of predefined strategies has been built into the ISMMS.

Those values which come from a fixed domain (i.e. decisions, priorities, the set of levels at which maintenance must be carried out) are selected orthogonally from the server allocation strategy. They may be calculated for each task either

1. **randomly**: fixed probabilities are assigned to acceptance or rejection for different classes of maintenance task at start-up time, and these probabilities are used to generate random values during execution.

2. **deterministically**: Acceptance is always the mandatory decision, the priority is ‘medium’ (2), and all the possible software levels are considered to be involved in every maintenance request.

Server allocation strategies may be selected independently of these modes.

### 4.3 Choosing Server Allocation Strategies

In the ISMMS prototype running in Automatic mode, the server allocation strategy selects which maintenance task is to be serviced in a given time slot, and determines the server that will provide the service. The strategy is executed repeatedly at each point in logical time at which it is required until no free and eligible server remains or no waiting and eligible maintenance task is unassigned, whichever occurs soonest.

Three allocation strategies are built-in, and may be selected:

1. randomly select the maintenance task to be served and randomly allocate an eligible server.

2. randomly select a maintenance subtask and allocate to it the most appropriate server – the available server with the greatest skill at the task, at that phase of processing.

3. select the most difficult waiting maintenance subtask for priority service and allocate to it the most appropriate server.

The above three server allocation strategies, together with the random and deterministic modes for decision-making and prioritization, give a total of six simulation strategies built into the ISMMS prototype operating in automatic mode. The user also has to set the logical time limit for which the simulation will continue to run in automatic mode. After this the ISMMS will return to the interactive mode of operation.

The ISMMS user can switch arbitrarily between interactive and automatic modes in one session, so the maintenance manager can manually drive the maintenance process into a given situation and then run different simulation strategies in order to test an allocation strategy and evaluate different allocation strategies under different conditions. The state is recorded after each interactive mode decision, and before and after each automatic mode sequence, so one can reliably go back again and again, change a parameter, and retest variant management strategies within the same scenario.

There are also two permanent display windows for the maintenance process state:

1. **graphic**: The length of the waiting queues are shown for each phase of the maintenance process. Maximum and prior values are also shown.

2. **text**: A detailed description of the contents of each queue is available, with task names, assigned priorities, and other characteristics.

At the end of a session (when every maintenance request has been either satisfied of rejected), the logical time consumed is returned, along with a statistical analysis of the session.
4.4 Setting up the Database

The ISMMS prototype includes databases storing information on (A) personnel skills, (B) toolsets characteristics, and (C) application software indices. These should be customized.

The personnel database contains records of all available servers of the maintenance department together with their ‘efficiency’. Each server has a certain efficiency at servicing a particular maintenance phase. This is an integer between 1 (lowest) and 5 (highest). As the maintenance process has many distinct phases, each server has an associated many-dimensional efficiency vector.

The application software database records the intrinsic difficulty of servicing that software in a particular maintenance phase and using a particular class of tools. The difficulty is an integer in the range 4 (lowest) to 20 (highest). If $E(s, p, g)$ is the efficiency of server $s$ for work on maintenance phase $p$ using a toolset of group $g$, and $D(p, g)$ is the intrinsic difficulty of maintenance phase $p$ using that group of tools, then the service time $S(s, p)$ needed by server $s$ to perform a service in that phase is:

$$S(s, p) = \min_g \text{round}\left(\frac{D(p, g)}{E(s, p, g) + 1}\right)$$

over all possible $g$, where

$$\text{round}(x) = \begin{cases} 
[x] & \text{if } x - [x] < 0.5 \\
[x] + 1 & \text{if } x - [x] \geq 0.5
\end{cases}$$

and ‘$[x]$’ is the integer value nearest to and no more than $x$.

Logical units of time can be easily transformed to physical units of time – the transformation is effected through multiplication by an adjusting factor, empirically determined.

Conclusions

This work has presented a model for the management of software maintenance which takes account of:

- Technical aspects of the software maintenance process.
- Metrics obtained on the software application being maintained.
- The organizations resources and requirements.
- The optimal use of resources.

The approach proposed has some restrictions: it considers, for example, that a software maintenance department is request driven.

Predicate Nets have been used to formalize the the model and make it suitable for implementation on a computer, the choice of Predicate Nets as a description formalism being influenced by its ability to express conditional transitions in distributed computations. Here that ability is used to direct synchronizations and express prioritization and scheduling constraints in a queueing network.

An Intelligent Software Maintenance Management System (ISMMS) prototype tool has been designed and implemented. The prototype is based on the software maintenance management model introduced here. It is written in the logic programming language Prolog, which has permitted fast prototyping and an accurate implementation of the Predicate Net description of the model.

The ISMMS prototype has been designed to assist the manager of a software maintenance department to allocate human resources to various incoming maintenance tasks. The results may well encourage further work in this area.
Further Work

The ISMMS tool has been built and designed under the REDO project\(^1\). Development is continuing and the final tool should have extended resource management capabilities over the prototype, and include management of more software and hardware resource characteristics in particular. Other components of the final tool should permit more flexible methods of customizing the tool to suit particular maintenance departments, and more interactive modification of the databases involved.

Customization of the software maintenance management model to the structure of a real software maintenance department will be a key issue in future.

References


---

\(^1\)Esprit II project number 2487; collaborating partners Lloyds Register of Shipping (UK), Oxford University (UK), University of Limerick (Ire), Marconi Command and Control Systems Ltd. (UK), Ingenieria y Tecnologia de Systemas, S.A. (Sp), Grumman Daten-Kommunikation, GmbH. (Ger), Electricité de France (Fr), Durham University (UK), Delft Hydraulics (Nth), Computer Technologies Company (Gr), Centrisa (Sp).