A Three-View Model for Performance Engineering of Concurrent Software

C. M. Woodside

Real-Time and Distributed Systems Group
Carleton University
Ottawa, Canada

Abstract

This paper describes a multi-view characterization of concurrent software and systems suitable for displaying and analyzing performance information. The views draw from well-known descriptions, and are compatible with established techniques and tools such as execution graphs, Petri Nets, StateCharts, structured design or object-oriented design, and various models for performance. The views are connected by means of a “Core Model” and are used together to extract information relating to system integration, such as interprocess overheads, and the delay behaviour of separate software components in complex systems. The integration of the views in the Core assists by converting results in one view (such as scheduling delay for resources) to parameters in another (such as delays along a path). The ultimate goal of the views is to support designers in making tradeoffs which involve performance, and to provide early assessment of the performance potential of software designs.

1 Introduction

To design a computer system for adequate performance requires tradeoffs between aspects of the system which are usually considered separately, such as the functional design or object design and its correctness, the processing load and the hardware capability together with its scheduling and management. Expertise and information tend to be compartmented between experts in these different domains, and information is difficult to transfer between them because of differences in definitions and methodology. It would be an advance to connect software performance engineering more closely to software design and resource scheduling, and this paper describes a notational and conceptual framework for making the connection. It is increasingly clear that more integrated approaches are needed throughout systems development. In concurrent systems, which may be either distributed or parallel, the challenges are greater and the pitfalls are deeper due to greater behavioural complexity of the software and the hardware. There are also rising expectations for quality, which includes adequate and predictable performance. There are many opportunities for moving information from one domain to another: performance models using early estimates of the processing cost of parts of a design can help in planning software architectures, in assessing the need for distribution of data and functionality, and in early hardware planning. There are also opportunities to do analyses that lie between the separate domains of expertise, such as hardware-software tradeoffs.

There is already an impressive array of methods and models for software development, proposed or in use. Some are rigorously mathematical, and some are embodied in CASE tools or performance tools. Rather than ignore them and start on a whole new model, this paper proposes a federated approach combining existing domain models. To explore what range of models can be included, a framework called Maps, Paths and Resources (MPR) has been developed specifically for software performance engineering. It includes a "Core Model" which is a kind of conceptual skeleton for the data describing the system, and three *views* (Map, Path and Resource) which provide the *View Models.* The role of the Core model is to capture the information in each view which is needed for performance engineering in the other views, and to support interview relationships. The role of each separate view is to support engineering analysis of some aspect of the system. Figure 1 illustrates the three-sided nature of the Core model and its relationship to the views. Each separate view is represented by a sketch using a typical graph notation, one of several which may be used for that view. The choice of the three views is consistent with many design techniques which use one or two views (but rarely all three); a commonplace example is structured design which is map-dominated but which includes path information in the data-flow/control-flow models, or some object-oriented design techniques which include path information as interaction diagrams or scenarios.

The three views are already widely used, in various forms. The *Map View* shows what is to be built. It shows the software *components* (processes, objects, modules, procedures) and their interfaces, data and invocation patterns. The sketch in Figure 1 shows processes as parallelograms, modules as rectangles and procedures as rectangles with a bar on one side; and is based on MachineCharts cite-buhr90. Other possibilities include structured design models cite-Yourdon2, object-oriented design models cite-ShlaerMellor92, and messaging diagrams. The analyses and decisions made in the map view concern modularity and the clustering of functions, and other concerns of design. The map view is related closely to the code. The *Path View* shows behaviour. It describes the *sequences* of events, actions and delays in the typical behaviour of the system. The sketch in Figure 1 shows a precedence-graph notation described further below. So many similar notations are used that none is widely accepted, but formal models for paths include partial-order definitions cite-pratt, execution graphs cite-CUS90, Petri Nets cite-peterson, and process models such as SDL cite-refSDL, CCS cite-CCS80 or CSP cite-CSP. The path view is closely related to scenarios, tests and traces of behaviour, use cases and response times. Each major response pattern has its own path definition. The *Resource View* describes the provision and scheduling of physical and logical resources. It includes the hardware (processors, storage, i/o devices, memory, interconnections) and logical resources such as locks or semaphores. Abstract models are also part of this view, such as queueing networks, timed Petri Nets, or simulation models cite-SES92. The resource view is related to contention delays and resource saturation at bottlenecks.

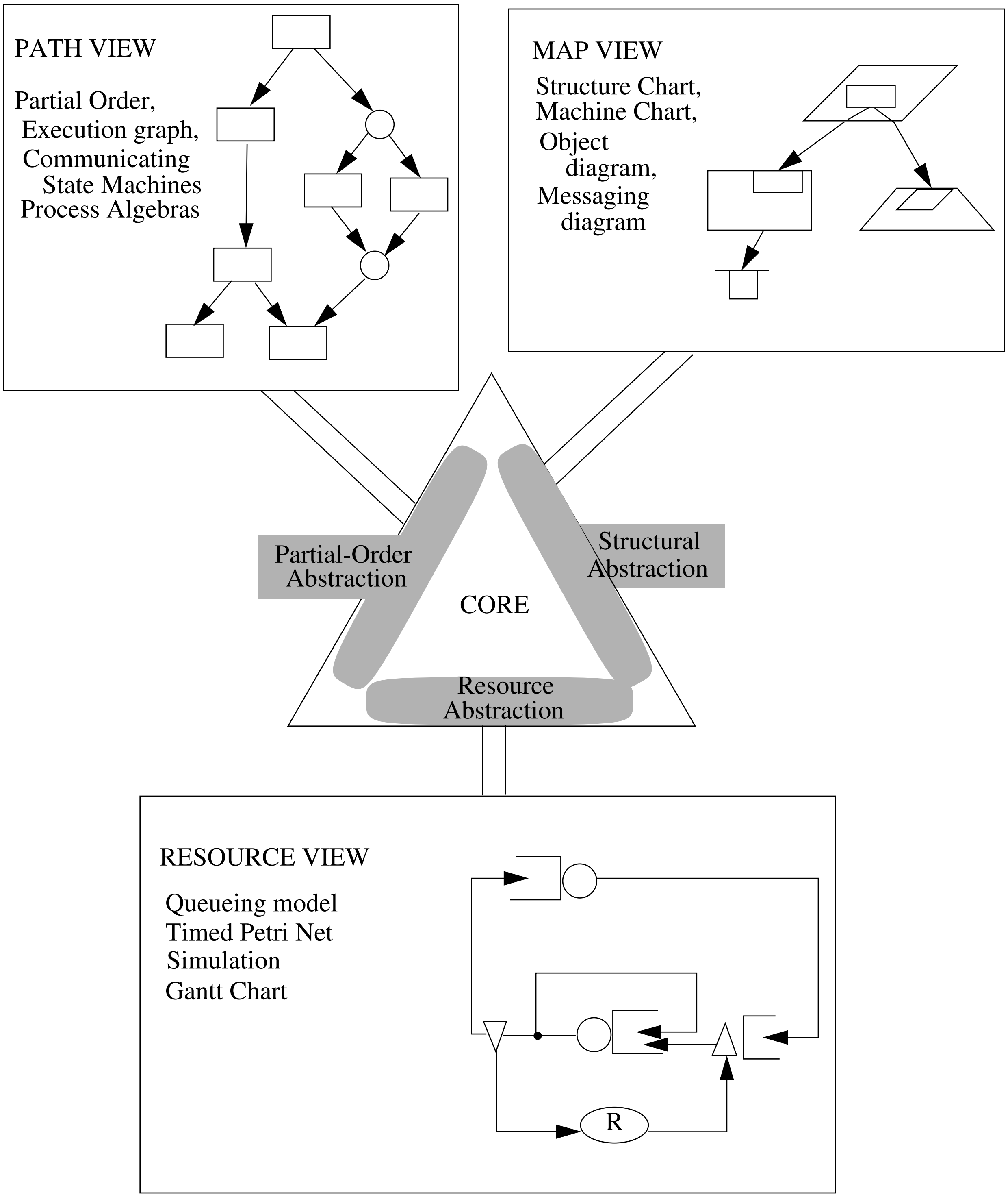


Figure 1: The Core Model and the Three View Models, with Candidate Models

Performance engineering demands that information should be exchanged between the views. The questions addressed in this paper are:

(1) what information does each view need to import from the others? This determines the characteristics of the Core model described in Section 3.

(2) how is the Core related to the typical view models? Well-known software and performance models are considered in Section 4, and a relationship to the Core is defined for each. It turns out that a key consideration is: how does the Core model deal with *abstraction* in a view model?

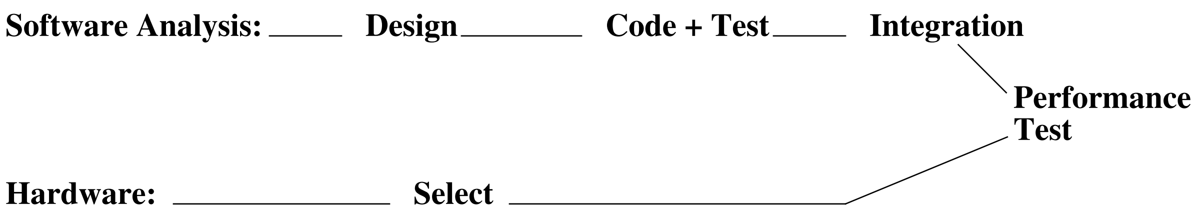
(3) how can one deal with incomplete information? A notion of *performance data coverage* (in the sense of including all the contributions made to load and delay by every aspect of the system) is defined in Section 4. The framework can sometimes supply approximations to fill gaps.

(4) a method for "pivoting" around a problem element, to see it within each view, is developed in Section 5 for solving performance problems.

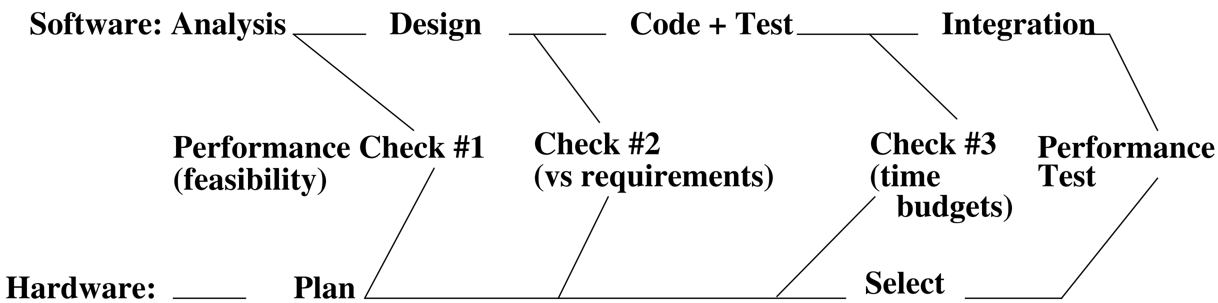
(5) How can the integration provided by the Core model be exploited to devise a performance engineering *process*? In Section 6, the MPR framework is used for a taxonomy of SPE processes, some of which are new. Using the framework one can create a process tailored to the needs of a given project.

The main goal of performance engineering is to improve the quality of software products. Usually the path information is considered only to a limited degree during the system analysis stage, after which the design (including the map definition) is developed from the analysis. The framework could strengthen the software development process. In practice, development often proceeds somewhat like Figure 2(a). No explicit model for performance is usually constructed, and performance is not examined until measurements are done during integration testing. Errors are found late, and may require significant design changes which substantially degrade the design quality and maintainability. Costly delays are not uncommon. What would be better is an early check on the probable performance, using planning values and models to fill in for the uncompleted aspects of the design. Figure 2(b) shows a series of performance checks based on increasingly complete information. The question is how to calculate these performance values at the early stages.

Previous performance engineering ideas have been used for resource analysis based on just one of the path view or the map view. Time budgeting and task graph analysis are two "path-dominated" approaches. *Time budgeting* allocates execution time to parts of a sequence in the path, and then tracks the implementation of those parts into components of the design; a component that uses too much can be detected in unit testing, and if high-risk components can be identified they can be designed and tested first. *Task graph analysis* carries this further, to complex paths with parallelism and synchronization and even to account for contention delays (e.g., cite-CHU91, [cite-PENG, cite-Thom). The weakness of both these approaches, which we can call "path-dominated", is that they are only indirectly linked to the decisions a software designer must make in terms of software structures such as tasks, modules and procedures. For single sequential programs several authors have shown how to reduce a path model to a workload model suitable for performance analysis; a good example is given by Smith cite-CUS90.



(a) Performance Evaluation in a Typical Development Process



(b) Performance Evaluation Earlier in the Process

Figure 2: Performance Evaluation within the Development Process

Alternatively one may try to analyze the design itself, which we may term a “map-dominated” approach. Some software design tools have incorporated some kind of performance evaluation (e.g. cite-SRVNseta90, cite-TBicse90, cite-HIT88, cite-SP93, cite-STP89) based frequently on simulation. Simulations generate paths, but often only in a hit-or-miss manner. Map-dominated methods would be improved by more systematic path analysis in two ways: more efficient evaluation focussed only on the known *most* important paths, and in using these important paths to guide the design of the components, for example to shorten a critical path. This is normally outside the scope of the tools.

Alone, the path-dominated and map-dominated approaches are deficient because each one lacks information available in the other. This paper points the way to a balanced combination, held together by the Core model mentioned above. The designer would be able to work in whichever view is most appropriate and can import information via the Core. It is in designing and maintaining concurrent systems that the greatest benefit may be expected.

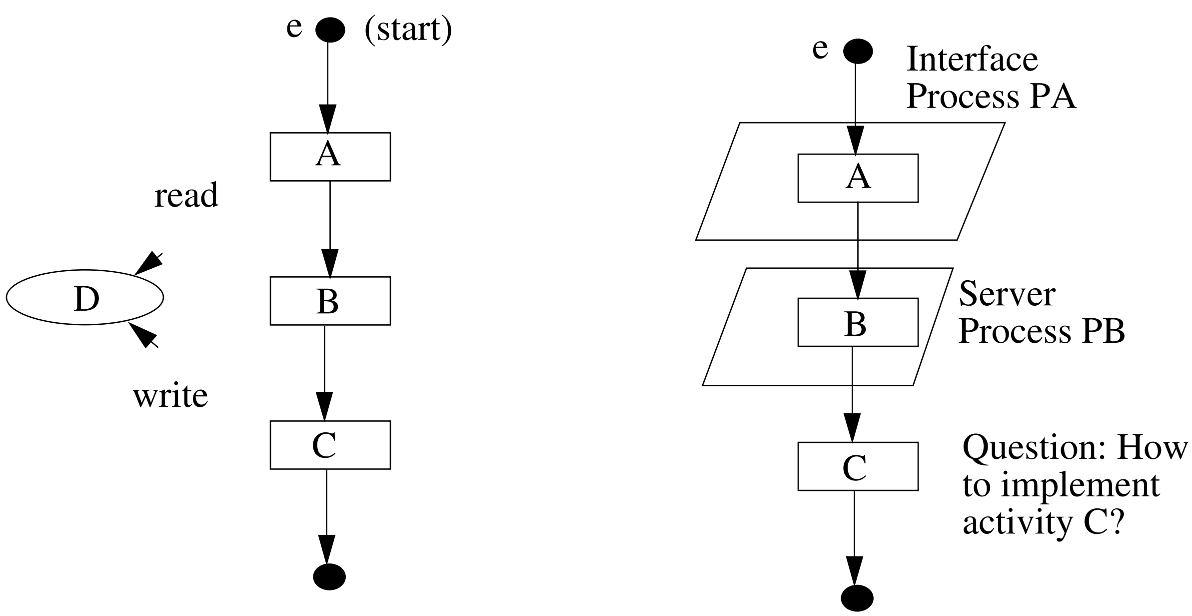
The list of references to existing techniques given here is necessarily incomplete, for very many separate and unconnected approaches have been described. It is believed however that the list is representative of the range of techniques, and that work not referenced here can also be integrated with the views and the Core model which will be described.

2 An Example

An example will be used to identify how performance issues may affect the larger concurrency structure of a software design. Consider a monitoring system with three activities *A, B, C* which must be carried out in sequence, as in Figure 3(a). *A* is a response to an interface event *e*, and is constrained to be executed on the interface processor, while *B* is a common service provided by an existing server process *PB*. *C* interprets the server output and writes data *D* which is accessed by later invocations of *A*. There will be many concurrent processes for *A*, each implementing the activity *A* for a separate interface, but only one server for *B*. Also, each interface has a stream of events *e* to deal with. Figure 3(b) is a partial design showing just one of the processes *PA* for *A,* the server *PB* for *B,* and the question as to how to implement *C.* Some alternative answers are explored in Figure 4, as follows:

Structure (a): Execute *C* as part of *PA,* as in Figure 4(a). *PA* sends a message to *PB* and then blocks to wait for a reply. Only when it has completed *C* can it accept another interface event.

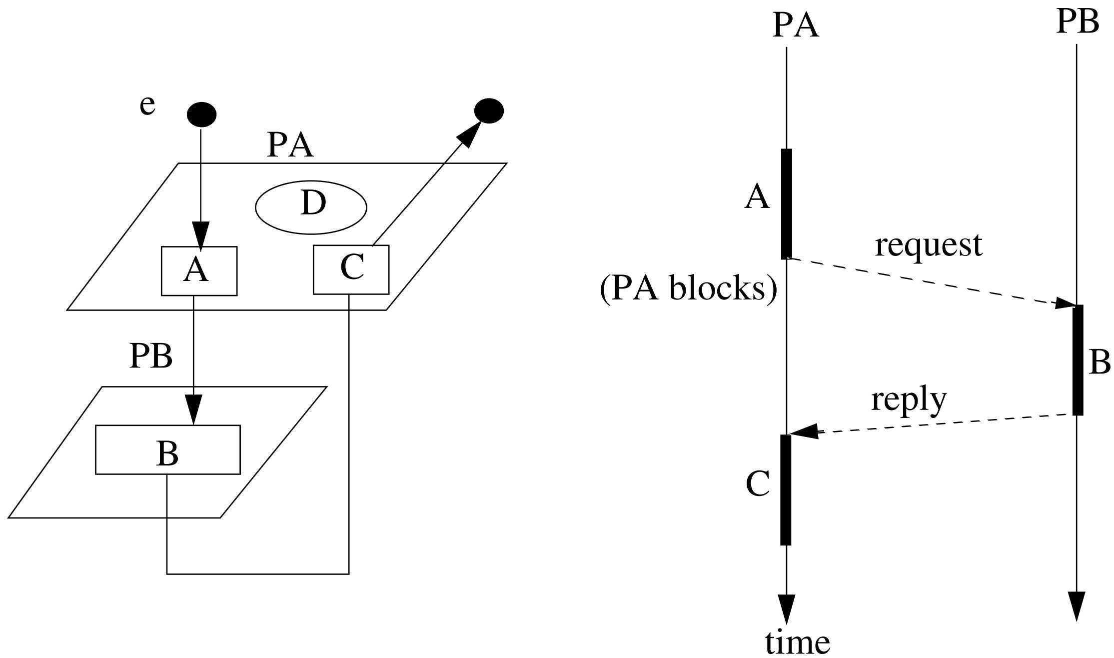
Structure (b): Based on *(a),* if another interface event may arrive and need processing before the previous one is disposed of then each interface can have several threads or replicas of *PA*, so each thread takes one event. Then some threads can be executing activities *A* or *C* while others are blocked waiting for the reply from *PB,* as shown in Figure 4(b). If the interface events are frequent and the blocking time is long (for instance due to a long network delay) and if the processor for *PA* is not too busy, a considerable number of threads might be used to advantage. Mutual exclusion on access to D by the different threads is necessary and is not shown. The Figure shows a timeline diagram on the right, with three threads labelled PA(1), PA(2), PA(3) (all associated with one interface) and the activities initiated by PA(*i*) labelled A(*i*), B(*i*), C(*i*).



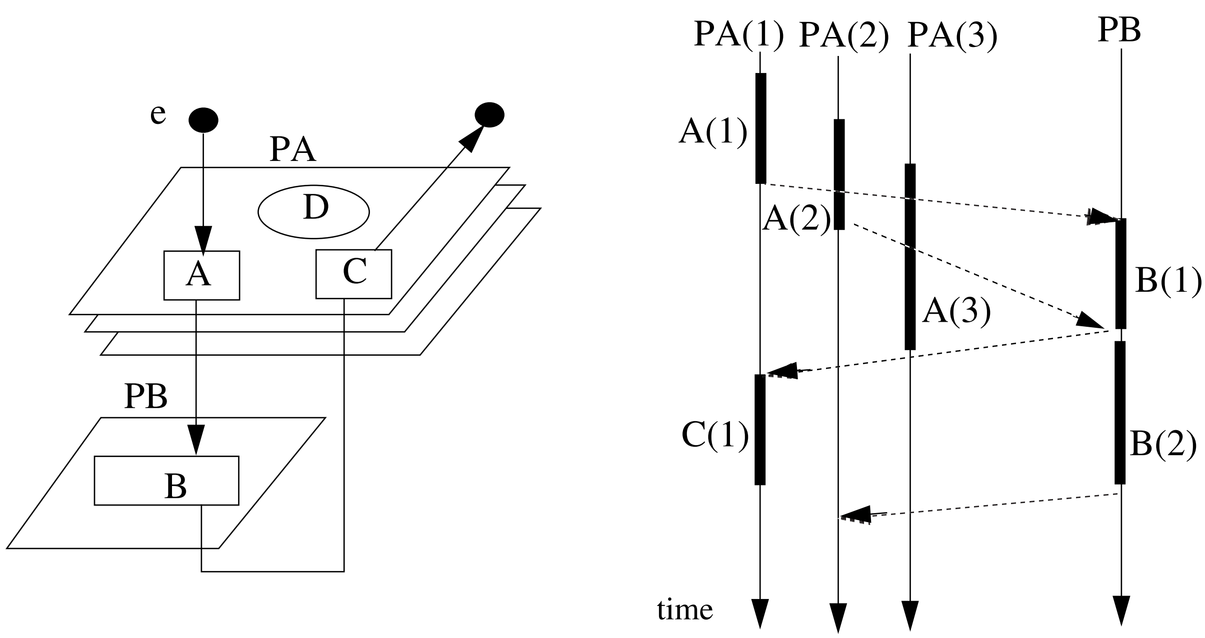
(a) The Sequence of Activities

(b) The Design Problem

Figure 3: A Small Example Design Problem

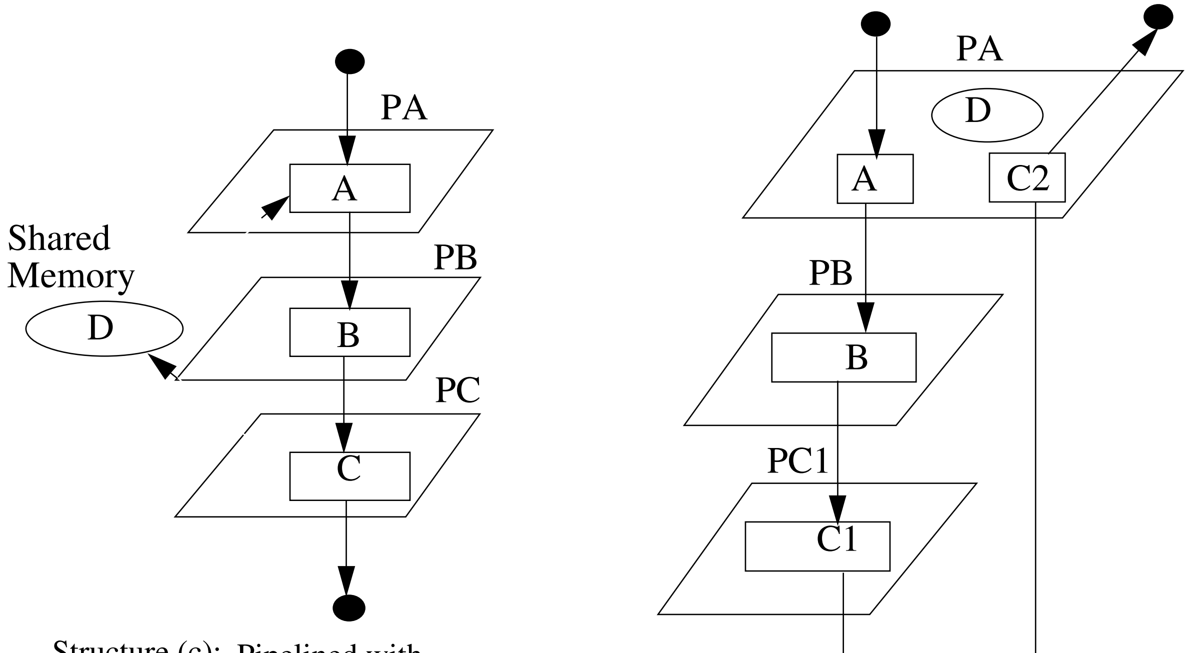


Structure (a): A Client-Server Structure



Structure (b): Multiple Clients and One Server for Each Interface

Figure 4: Architectural Designs for the Problem of 3.



Structure (c): Pipelined with Shared Memory

Structure (d): Pipelined, with the Result Forwarded

Figure 4 (cont'd): Architectural Designs for the Problem of 3.

Structure (c): Execute *C* in a separate process *PC,* as in Figure 4(c). If some of the processors running *PA* are heavily loaded, *PC* for those cases can then be located on a separate processor to reduce the delays and increase the capacity, however the two processors would have to share the memory for *D.*

Structure (d): In a system which does not have shared memory we can still unload the processor for *PA* if *C* is executed in a separate process *PC* on a separate processor, but it sends the resulting data in a message to *PA* (as in Figure 4(d)). This requires adding an activity *C2* to *PA* to receive and store the data.

To make the best choice among these four alternatives one must evaluate their throughput and delay effects. This requires knowing or predicting the values for a substantial set of performance data, beginning with execution times, overheads, the frequency of invocations, and the number of interfaces. It may be preferable to use ranges of values, rather than point values, to represent the range of expected situations over the life of the software. Other, non-software choices must also be represented -- the number of processors available, and the scheduling priorities of processes, for instance, affect the performance. Finally, other competing workloads on the same processors cannot be ignored as they may cause delays to the processes being designed.

The outstanding difficulty of design problems like these is the meshing of the issues of processing sequence, processor structure, execution time and frequency, competing workload, and processor resources and scheduling.

How is the performance comparison to be computed? The first step is to assemble the information from our three views. This is often cited as the major barrier to early consideration of performance, as the information may be scattered or inconsistent (including data from manufacturers, measurements on earlier systems, estimates made during analysis, details buried in design documents or in designers' unrecorded decisions) and may even be missing. Smith has described a systematic process for obtaining subjective estimates of missing parameter values cite-CUS90. This highlights the fact that software performance is *first of all an information management problem.* The main purpose of the Core model is to help marshal the information.

Second, a performance model must be built and solved. This is a challenge to the capabilities of modelling technology. For systems of significant size to be solvable, the models must suppress detail strategically; the problem is typically known as “state explosion”. It is clearly visible in an obvious direct approach which is not advocated here, which may be called the “big flat transition model”. All the “important” states of the software system are identified in the design model, using state-transition models for the separate processes and data items. These are then composed together, and further states are added for the network, the processor schedulers and to describe the generation of external events. Time is added to all these states using processing times, network delays, event rates, etc. and a large Markov chain model is obtained which (in principle) gives the performance. If the model is too large to solve it is decomposed and approximated until a feasible solution is found. The attractive feature of this approach is the clear handover from description of the logical flow in the transition system to performance modelling, at the point of introducing time. Its defect is in the truly vast size of the state spaces - it is easy to imagine a system with 10^{100} or even 10^{1000} states! Although this direct approach is mathematically sound, practical models must use approximations.

We wish to consider “design-directed approximations” such as Rendezvous Networks cite-SRVN as performance models. They use decomposition and approximations based on the structural and behavioral components of the software and the hardware. Therefore they require much richer information about the software description than is contained in the “big flat transition model”. The three views introduced above can supply the information. For example, the performance solver can retain the structure of a particular fork-join in the path description, and use the knowledge to calculate the delays along the separate paths within the fork-join, in order to find the synchronization delay at the join. We will consider the information necessary to do design-directed approximation and how the Core Model can provide it.

3 A Definition of the Core Model

The definition of the Core model will be derived from the nature of the information which is normally defined in one view, but is also needed in others. Since the information needs of the resource view are the most important for performance evaluation, they will be considered first.

The *resource view* must have a description of the software workload, and the simplest description is the average workload. This gives just the total execution demand for each resource and each response (i.e., for each major path), aggregating out and ignoring all information about the sequence of demands. It provides the information required by queueing network performance models, as used for instance in cite-CUS90. However, we will require a fuller description, because in systems with real parallelism and synchronization the full detailed execution graph must be known. It may be used directly as in [cite-PENG or may be reduced in solving the model, as in cite-HEIDTRIV83, cite-Thom. It is needed even when there is only one processor and no real parallelism, if there are many concurrent competing programs. Therefore, the Core model will include the following execution-graph or activity-sequence information obtained from the path model:

– a set of activities A, with an execution demand, a repetition count and resource requirements for each one,

– three events associated with each activity,
  •“ready(A)”, when activity A is ready to begin executing,
  •“begin(A)”, when it does begin,
  •“end(A)”, when it is finished.

– a set of predecessor and synchronization relationships defined on the events “end” and “ready”, which establish a partial-order model for the path.
  •Direct sequence, and and-forks and and-joins, will be specified by relationships of the form Sync(in-list; out-list, number) defining a “sync-point”. In-list is a set of end-events needed to trigger the sync-point, which then generates the events in the out-list. “Number” is optional and may be used to specify a smaller number of events in the in-list which must occur before the events in the out-list are triggered (in this case the remaining in-events are ignored). For standard fork-joins “number” is the length of the in-list of events.
  •Or-forks and or-joins (which represent for example case statements) are specified by a relationship Choose(in-list; out-list, prob-list) defining a choice-point. One in-event from the in-list causes one out-event from the out-list to occur, with the probability given by the corresponding item in “prob-list”.

The definitions above implicitly also define two subsets of events:

– a set of path-begin-events for each path, being ready-events with no predecessors,

– a set of path-end-events, being end-events with no successors.

This basic path information provides the foundation for the Core model. The usual way to visualize this information is as a directed graph with three kinds of nodes, activities, choice points and sync-points, as shown in Figure 5.

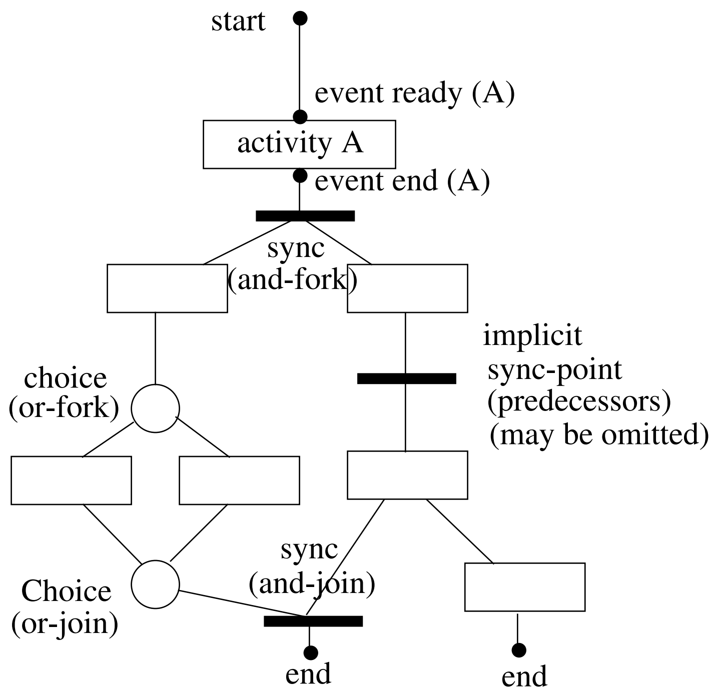


Figure 5: Notation for Paths in the Core Model

The resource view must also be informed of the resource requirements of each activity, which can include the processor it runs on, the process instance which executes it, a passive module that implements it (which may be identified with the process, or not), and semaphores, critical sections which include the activity, and data resources. Most of these are aspects of the design, which places them in the map view. (Path models also specify resources for activities, so consistency must be checked). In any case, they need to be imported to the resource view via the Core model. The following definitions are suggested as generic representations of the map information:

– a set of processes,

– a set of modules, which may be nested, and interfaces for each module,

– calling relationships from modules to interfaces of other modules,

– a set of logical resources representing semaphores, critical sections, and locks on data, with relationships identifying the modules which use them.

A process represents an independent thread of control, and a scope for data (as for an Ada task). Modules may be designed to be active or passive. Although it is not part of the software description, we will also include in the map the intended set of processors and their interconnection, as in concurrent systems this is often an integral part of the design.

Finally the resource view has information local to itself, which includes scheduling disciplines for logical and physical resources, allocation of processes to hardware, and parameters for resource speeds and multiplicity.

Now we can examine the information needs of the *path view*. It is natural to define a response time in this view, as the sum of the times for activities along a particular path. For response time calculations the path view needs to import the delays due to resource contention, which are calculated in the resource view. Contention can be represented by:

– an “expanded” value for the time of each activity, including its share of the total waiting for the resources it needs. The expanded time represents the delay between the ready event and the end event of the activity, which are already part of the Core model.

If the path view is to describe the execution time correctly it also should capture the overhead execution caused by the design. In general, overhead occurs on crossing map boundaries, such as in making procedure calls, for interprocess messages, or for context switches between processes. Special *overhead activities* can be added to the path model at the boundary crossings as defined in the map, or the overhead time can be added to the existing activity times. In either case the Core model already conveys the necessary activity-to-map correspondence. Some overhead execution is conditional on the sequence of events, for example if a message is waiting when a process executes a “message receive” command then there is usually no context switch required. The path model can resolve the probability, given the contention expansions. Then this probability can be attached to the overhead activities in computing path times. For this reason the Core will include:

– *potential* context switch and messaging overheads, represented as special activities, with a probability of actually having to execute each one. The probability must be calculated from partial path times.

Finally we can consider what information should flow into the *map model*. The amount of time spent executing a particular program component, in the form of a percentage of total time, is often required for performance tuning. To compute this we need to know the set of activities mapped into the module (already defined), the execution demand of each (defined), and the frequency of execution of each activity. The Core already includes the activity frequency relative to the path execution itself, so in addition we only require:

– the path (or response) frequency, usually called the *throughput* for that particular path. This is the rate of occurrence of the event that begins the path.

The throughput is determined either in the resource or the path view. If it is a free-running path, which restarts after each completion, the resource model must be solved to get the throughput and it comes from there; if the path is driven from outside at a known rate, the rate may be attached to the event that begins the path. In any case the Core model will include the throughput for each path.

This completes the most obvious information needs of a basic Core model. The development has also shown how the Core allows each view model to be completed, and to provide information to the other views. The nature of the interfaces between the Core and the separate view models can also be inferred. The next section will examine this question of interfaces in more detail for certain commonly used view models.

The flows of information between views for various performance-engineering functions are now identified. For example, to determine the overhead and its effects on performance, information must flow from the path view (call it P) to the map (call it M), back to P and then to the resource view (call it R). That is, sequence information in P identifies border crossings in M, which gives overhead activities in P, which become workload items in R. We will call this a P-M-P-R process. To incorporate the contention solution into the probability of executing an overhead activity, a loop back from R to P to update the activity delays with their expanded values, and an iteration P-R-P-R..., can be used. In general, a demand-based strategy can be used to find information needed in one view by first querying the Core for it, and then if necessary going to another view and carrying out necessary operations to obtain updated values. The different ways in which the views may be combined is the subject of the taxonomy given in Section 6.

4 Typical View Models

Most well-known software models fit into the MPR framework, and can be interfaced to the Core model defined in the previous section. This section will show how some models fit each view, remembering that it is not the purpose here to define new models but to integrate the ones we have.

Path Models

Any model which describes the ordering and causality of events can be used as a path model. The list is long, going back at least to Karp and Miller cite-KARPMIL nearly thirty years ago. We can distinguish three classes of models: partial orderings and graph-based models, structured models based on state-machines, and process-algebraic models. The semantics of concurrency differ in some cases, and it is already clear from the definition of the Core that the partial-order semantics are given primacy in this research. However the use of the interleaving semantics of the process algebra approach seems to be straightforward. Data flow models may also be candidates for path models, if they have an adequate execution semantics. The original data-flow definitions did not, but the later extensions by Hatley and Pirbai cite-HATLEY], for instance, do.

The partial-order and graph-based models include Smith's execution graphs, the task graphs used by Chu and others, and Petri nets. Several authors have used versions of Petri nets as a description of concurrent software problems, including Yau and Caglayan cite-YAUCAG83, and Peng and Shin [cite-PENG. DesignCPN cite-designCPN is a tool for defining programs in a Petri net framework. We will describe how a view model using high-level Petri Nets corresponds to the Core model, using Figure 6. In the Petri net an activity is represented by a high-level transition (a box), executed by a token which arrives at a place (a circle) which is connected to the box. An end-activity event is the completion of “firing” of the activity transition, while a ready-event is any firing of a transition which puts a token in a place which enables an activity transition. A synchronization involving more than one input or output is a simple untimed transition (a bar). A choice is a place (circle) connected to a transition for each alternative output choice, and simple sequence, which is a synchronization in the Core, is also represented by a place with one input and one output. Loops are identified as such in the Core model, but not in Petri nets; rather they are described implicitly by looping the path back on itself. To interface to the Core, an operation is needed to extract loops and an interface operation to let the user define the loop count. If the loop count is defined by a probability of looping back then the average count is easily obtained.

Any practical use of a path representation demands the use of abstraction, meaning the ability to hide complex subpaths within a single activity, or to refine an activity into a sub-path. This is partly because it is impossible to know, when developing a design, what the primitive activities are, and because abstractions provide a suitable level of description for a given purpose. All the various path models employ abstraction; for example, in high-level Petri nets a single-input, single-output subnet can be "hidden" in a high-level transition. Different levels of abstraction in the path model sometimes correspond to levels of abstraction in the map or resource models too – this point will be examined later.

The second class of path models uses a set of communicating state machines, as in SDL cite-refSDL or StateCharts cite-HAREL87, or the control flow model of Hatley and Pirbai cite-HATLEY]. In this case the sequences we need are given by the set of traces of the system. However this style of definition tends to generate a great number of paths, many of which are important for correct behaviour but occur relatively rarely, and so can be ignored in performance analysis. To identify the remaining relatively small set of "important" traces of a state-machine model, it must either be augmented by the set of important traces itself, or by selection rules that allow it to be determined. The main observation to be made is that the set of traces is essentially another graph model for the sequences.

Considering abstraction in state-machine models, activity abstraction (in our sense) is provided by StateCharts, in which a state may represent an activity, and may be refined to a state machine, or to a set of concurrent states which may in turn be a set of state machines. Many other state-machine models however do not explicitly support abstraction.

The third class of path models use process algebras, such as CCS cite-CCS80, CSP cite-CSP, and LOTOS cite-LOTOSdef to define event sequences. A process is a high-level abstract activity, and the algebra in each case provides a notation for composing subprocesses, or refining a process in terms of lower-level processes, with various combinators. Activity abstraction is therefore central to the approach. Here again the sequences we need are the set of traces, and again there is an advantage in being able to discard traces which are not central in defining the performance. If we use the traces themselves, then the basic path definition for manipulation and interfacing to the Core appears once again to be a graph model.

The use of a set of event/activity traces for a path model is a kind of last resort, and more compact representations which represent choices among alternatives explicitly are also possible in most cases.

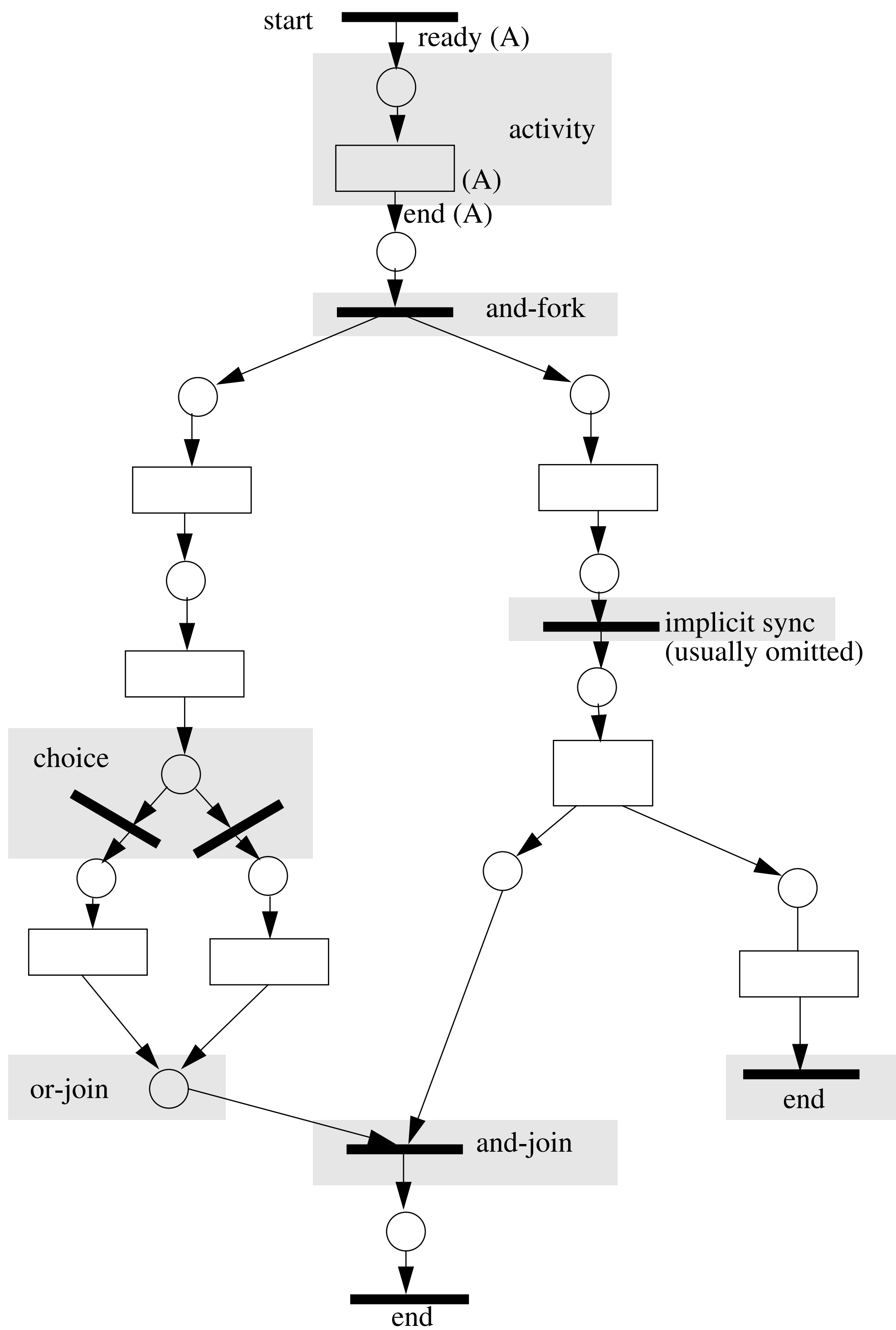


Figure 6: Petri-Net Equivalent to the Core Path Model in 5, with Notational Correspondences

Map Models

The best known map model is the software structure chart cite-Yourdon2. The basic form is a graph in which nodes are modules and directed arcs represent calls or transfers of control from one module to another. Application modules at the “top” of the graph execute overall control and call modules providing services, which in turn call for lower level services. The higher level modules are in this sense at a higher level of abstraction. The structure chart can be generalized to represent modules with nested and hidden internals, as in Ada packages, or in modules in Modula. Then there may a number of distinct interfaces in a single module, for accessing different services, and the interface components provide a “top” level above layers of internal hidden modules which provide details of these services. Finally if there are concurrent processes they are like modules, but they are active in that they also sustain a separate identified thread of control. MachineCharts cite-buhr90 provides a notation like that in Figure 7 for these essential elements. It also shows the use of passive resources as “reactors”. Figure 4 used parts of this notation for the example. The Core model can extract from a MachineChart the modules and resources and the location of each module in the nesting hierarchy, and in the calling graph within its surrounding module. The concurrent processes and other software resources are identified.

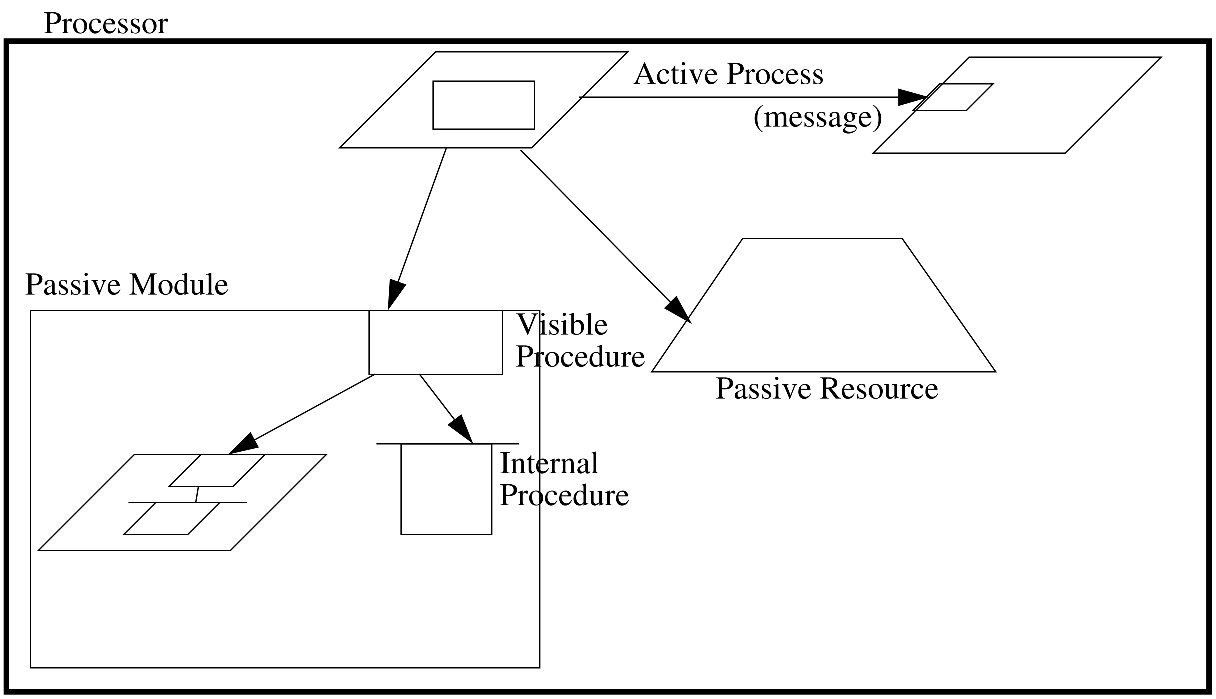


Figure 7: A Sample of Notation for the Map View, based on MachineCharts.

Activities and software components are both defined at several levels of abstraction and are cross-referenced. A high-level activity may be spread over many components, and a component may execute many activities. However by going to a fine enough level of activity detail, we can always find a set of activities which are wholly executed by a given module. It will be applied as a principle for defining the allocation of activities to modules, that the activites must be sufficiently refined. A good result of this is that whenever the execution crosses a component boundary it also generates one or more activity-model events, so the one set of events covers map-view behaviour too.

For concurrent and object-oriented systems a commonly used (but limited) map description is the message diagram, showing the interchange of types of messages between two or more objects or processes, during some defined operation. Figure 4 includes message diagrams for structures (a) and (b). Message diagrams illustrate the intention of this paper, because they explain the desired system behaviour by combining the path aspects of the message sequence and the map aspects of location of processing between the messages. These diagrams are widely used and understood, and are in line with the aim of this paper, to build more inter-viewpoint connections at more detailed levels of abstraction. The map view given here is more informative, but a slice through it at the process level gives a message diagram.

Resource Models

Resource models represent the flow of demands for resources (the workload) and the rules for dispatching or scheduling the resources, and have means for calculating performance figures as delays, throughputs and utilization. The basic unit of demand in our terms is the activity, and the workload description is based on the flow of execution of activities. The resources are partly identified in the map (e.g. single-threaded processes), partly in map-to-resource correspondences (e.g. process-to-processor allocations) and path-to-resource correspondences (e.g. logical resources like locks in a database), and partly in the resource view alone (e.g. buses joining processors). Certain types of model are suitable for special combinations of workload and scheduling discipline. In the best-known model, the *queueing network*, the workload must be described as a set of non-interacting threads of control (i.e. paths) which use one resource at a time. There are many texts on queueing models (e.g. cite-LZGS, cite-JAIN92) and many computer packages for solving them cite-QNAP2, cite-macnairsauer. Extensions have been developed to deal with more complicated workloads, in which paths may fork and join, and activities may require multiple resources held simultaneously. There is a large literature which cannot be dissected here, but it is worth mentioning de Souza e Silva et al, on extensions to queueing methods cite-deSouSilv1, Thomasian and Bay cite-Thom and Chu et al on combining task graphs with queueing cite-CHU91, and the models for client-server systems in cite-SRVN and cite-SRVN89.

A distinct thread in resource modelling has been based on detailed finite-state representation of the control flow (combining path and map elements) and the resources. Timed Petri Nets have been found to be suitable for this, and the work of Chiola et al is representative cite-australia.

Design-Directed Decomposition

Most of the resource models decompose the complete system model into submodels to make it practical to solve. The criteria for decomposition are often visible in the design, because the same criteria of cohesion for including activities and resources into a design component also suggest the submodel structure. Components are included in the same submodel if they use much the same resources, are tightly coupled together, and operate on roughly the same timescale. Peer submodels (on the same level in the hierarchy) operate on similar timescales and communicate with each other, but are less tightly coupled, and use mostly different resources. Timescale differences create a vertical hierarchy of models, with subsystems that operate on faster timescales placed at a lower level in the hierarchy. They may be identified by the small activity times in subpaths which traverse them. Sometimes the rules can be applied in a simple and obvious way. For example, in a distributed database system it may be proper to model the individual database nodes as peer submodels, with the linking network as one more submodel. Within the network there might be a lower level submodel for each link; within each database there could be lower level submodels for the lock manager and for the disk subsystems.

Design-directed decomposition exploits knowledge of the path and map. In cite-CHU91, a partly aggregated version of the path constitutes a higher level model, which can be solved if activity delays are known. These are found (including resource contention delays) by a lower level model which represents the activities and processors. In cite-SRVN a submodel is constructed for each software process, based on the activity demands associated with it. Generally the specification of the design and behaviour is useful in making effective decompositions. Further, the design components are often a suitable basis for monitoring performance (using events such as message passing or procedure calls), to give measurements that can be identified with elements of the other views.

Data Coverage of the Three Views

Incomplete performance data is a stumbling block in many projects. One reason is that data must be obtained in one view and then used to compute a value in another view. For example a profiler gives the total execution time for a procedure during a test, and the number of calls. The time can be divided by the number of calls and copied into the path model as the time for the activity implemented by that procedure (provided there is an exact match). The number of tests performed would have to be enough to cover all the activities in the paths, and to provide data inputs which give representative activity times.

Clearly the Core model can assist in filling in the data values, by establishing the correspondences and the parameter relationships, and by listing the parameter values which are needed. Coverage metrics are easily defined based both on the simple availability of values for parameters (such as activity times, processor speeds, or context-switch overheads) regardless of their accuracy, or as a measure of quality of the values we have.

The Core model can also be used to estimate parameters which cannot yet be measured, by extrapolating from available data. Table 1 summarizes data which is typically available in each view and data which can be obtained by using other views.

Table 1: Performance Data Coverage at Design Time

Data typically available in each view, separately		
Map View	Path View	Resource View
1. Load, as execution time per invocation for some components, (scaled from the values for a previous implementation of a closely related function).	1. Activity execution times, which may be estimated by path analysis [Smith]	1. Total processor loading per response, for a previous closely related system
2. Overhead per invocation for all components		2. Typical resource utilizations, for previous systems
Typical Important Additions, Using the Other Views		
3. Load breakdown by activities determines loadings per invocation for different alternative component designs	2. Contention delays may be estimated, for each activity	3. Processor loadings per response for different allocations and different software architectures
	3. Overhead times can be factored into paths	

5 The Cooperative Use of the Three Views for Design

The reasoning needed to develop a solution to a performance problem can benefit from the cooperative use of all three views. For example, when a response time is too long the primary factors are the activities which must be completed in the response, which are visible in the path view. However a contributing cause might be an extra delay due to contention for a resource somewhere along the way, and this contention will only be seen in evidence from a measurement or model in the resource view. Candidate solutions may be associated with any of the views. Examples of candidate solutions, labelled M, P or R according to the view in which they are introduced, are:

- more or faster processors, buses or memories (R)
- use of priorities (R)
- replication of logical resources (R)
- tighter code or more efficient algorithms in a critical area (M and P)
- restructuring of the processes (M)
- caching of data to reduce storage accesses (M)

- fewer separate processes (to reduce overhead) (M)
- exploit potential parallelism between activities (P).

Even though a solution may mainly involve one view, to choose it and evaluate it involves all the views. This section will look at methods for doing so.

Pivoting Between Views

Using the Core Model we can go from a "problem" element in one view, to obtain perspectives based on that element but within the other views. Figure 8 expresses this as a shifting of perspective into different directions surrounding a particular component *m*. It includes a perspective within *M* showing the relationship of *m* to the system as a whole. We will call this "pivoting" between views, based on a "pivot element" and the corresponding elements in the other two views, as follows:

Pivot Element	Corresponding Elements in Other Views
activity *a* (in *P*)	$\boldsymbol{M}_A(a)$ = component(s) that implement *a*
	$\boldsymbol{R}_A(a)$ = resources used by *a*
Component *m* (in *M*)	$\boldsymbol{A}_M(m)$ = activity(ies) implemented by *m*
	$\boldsymbol{R}^*(m) = \boldsymbol{R}_M(m) \cup \boldsymbol{R}_A(\boldsymbol{A}_M(m))$, where: $\boldsymbol{R}_M(m)$ = resources used directly by *m* $\boldsymbol{R}_A(\boldsymbol{A}_M(m))$ = resources used by activities in $\boldsymbol{A}_M(m)$, (e.g. logical resources)
resource *r* (in *R*)	$\boldsymbol{A}_R(r)$ = activities that use *r*
	$\boldsymbol{M}_R^*(r) = \boldsymbol{M}_R(r) \cup \boldsymbol{M}_A(\boldsymbol{A}(r))$, where $\boldsymbol{M}_R(r)$ = components that use *r* directly $\boldsymbol{M}_A(\boldsymbol{A}(r))$ = components that implement activities that use *r*

The correspondence between resources and components (between *R* and *M*) takes account of the fact that components may be mapped directly to resources (as when a process is allocated to a processor) or may have an indirect dynamic dependence through an activity which requires a logical resource such as a lock. The sets $\boldsymbol{M}^*(r)$ and $\boldsymbol{R}^*(m)$ combine both the direct and indirect dependencies.

By pivoting on a problem element we can evaluate the problem in all possible ways and identify more potential solutions, than if we stick to a single view. We will consider two examples based on the set of alternative designs shown in Figure 4, and we will use a multi-view "pivot-table" to display key information.

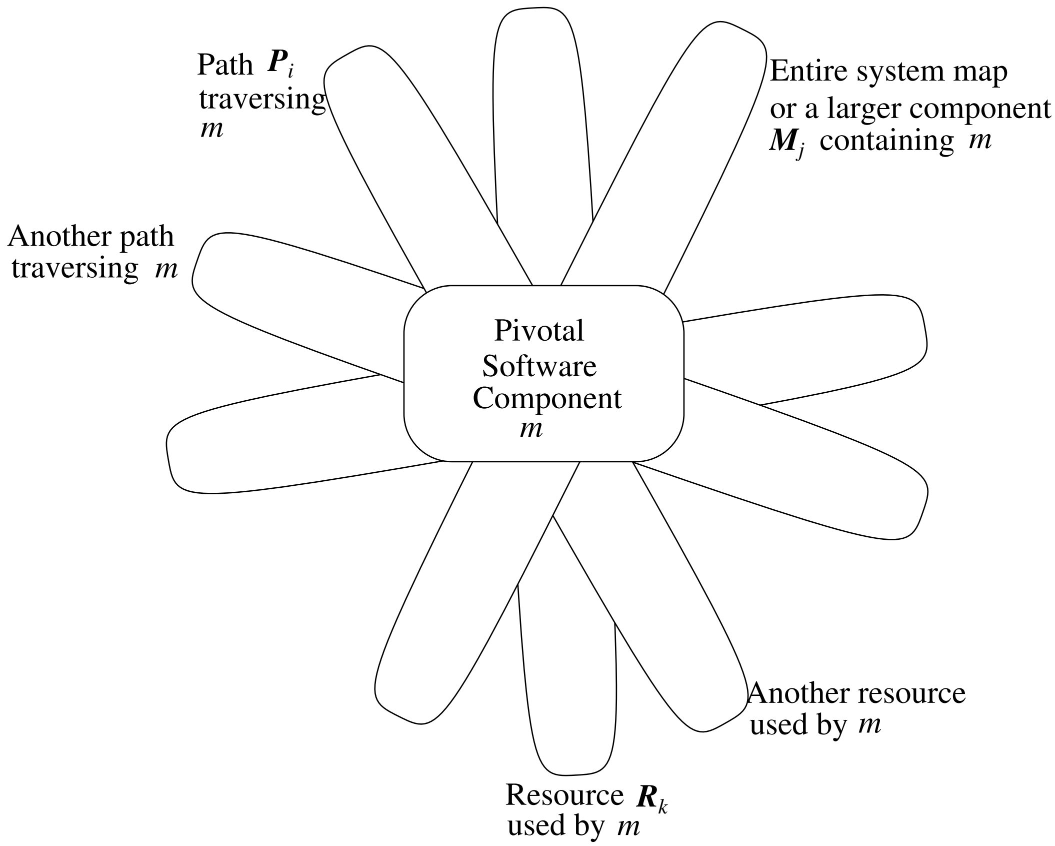


Figure 8: Path, Map and Resource Perspectives for a Pivotal Element (here the pivotal element is a map element *m*).

Example (a): Congestion and Resource Replication: Consider the design of Figure 4(d), and a situation in which the time requirement to complete the path A-B-C1-C2 cannot be met because PC is overloaded, introducing waiting delays. There are 5 interfaces, each with a process PA and each triggered at intervals of 3 seconds, giving a total path frequency of 1.67 inputs per second. PA executes the first activity A with t_A = 1 sec and the last activity C2 with t_{C2} = 0.1 sec, and is blocked (held busy) in between. C2 must complete within 3 seconds of the input if PA is to keep up. There is one server PB (with t_B = 0.2 sec) and one server PC1 (t_{C1} = 1 sec) serving all 5 streams. The completion time of 3 seconds is comfortably greater than the total computing for each response at 2.2 sec. However the single server PC1 cannot keep up with the five streams, which offer it a load at 167% of its capacity, because the implementation of C1 has turned out to be more demanding than expected. Table 2 shows the processing time resource demands for the activities, and the available capacity in seconds of processing per second, in the original situation. To run this system as it is the interface event rate has to be reduced.

A Markovian performance model was constructed assuming (for simplicity) that the activity times are random with an exponential distribution, and solved for the reduced event rates. At a six-second triggering interval the busy time of PA including blocking is 4.4 sec and the response delay to a triggered event, including waiting for PA, is 7.0 sec; at a 15-second interval these delays are 2.81 and 2.82 sec.

The pivot table (Table 2) has side-by-side lists of elements in the three views, at some suitable level of abstraction for each view. The pivot is activity C1 (shaded) which is overloaded, as discussed above. The other bold-faced entries are the corresponding elements - here there is just one of each, but there could be a set in each column. The main performance indicator for each element is shown - load per response for components, utilization for resources (at the stated frequency of 1.67 responses/sec.) and time per response for activities. The table reveals clearly that the problem is in the resource HC - it is overloaded at the given response rate.

Table 2: Pivot Table for Example (a) at a Path Frequency of 1.67/sec

Map View		Resource View			Path View	
Process	Load (sec/resp)	Processor	Multiplicity	Utilization	Activity	Time
PA	1.1	HA	5	37%	A	1.0
PB	0.2	HB	1	33%	B	0.2
PC1	**1.0**	**HC**	**1**	**167%**	**C1**	**1.0**
					C2	0.1

To deal with the desired event rate, we will consider increasing the resources for activity C1 by providing n_C copies of PC1 and HC1, and a dispatcher which selects an idle copy. The dispatcher will be co-located with PB, since it is used immediately after completing activity B. Figure 9 shows the modified activity sequence overlaid on the processes. It executes a "dispatcher overhead" activity DOH with the time demand $t_{DOH} = 0.1$. Table 3 shows the modified demands and capacities, and the last column shows that if n_C is 2 or more then server PC1 will have adequate capacity (its utilization will drop below 100%). Also the dispatcher has increased the utilization at the processor for PB, which may cause concern about delays there. We will also provide a queue for events waiting to be processed by PA at each interface.

A Markovian performance model was constructed for the modified case with 1 to 5 copies of PC1. To simplify the analysis the number of events waiting for processing by PA was limited. The performance results are shown in Table 4 with the time to execute each activity. Activities B and C1 include waiting for PB and PC1. Once PA is dispatched there is no additional waiting for activities A and C2, and the first total gives this time. The second total includes waiting to dispatch PA and is the response time to an event. From the performance results in Table 4 we see that n_c=2 is not enough, for contention delays at PB make the queue at PA only marginally stable. With n_c=5 the system can keep up but responses are completed only 4.56 sec. after the triggering events because of queueing delays before PA begins to process each event. This may require further improvement.

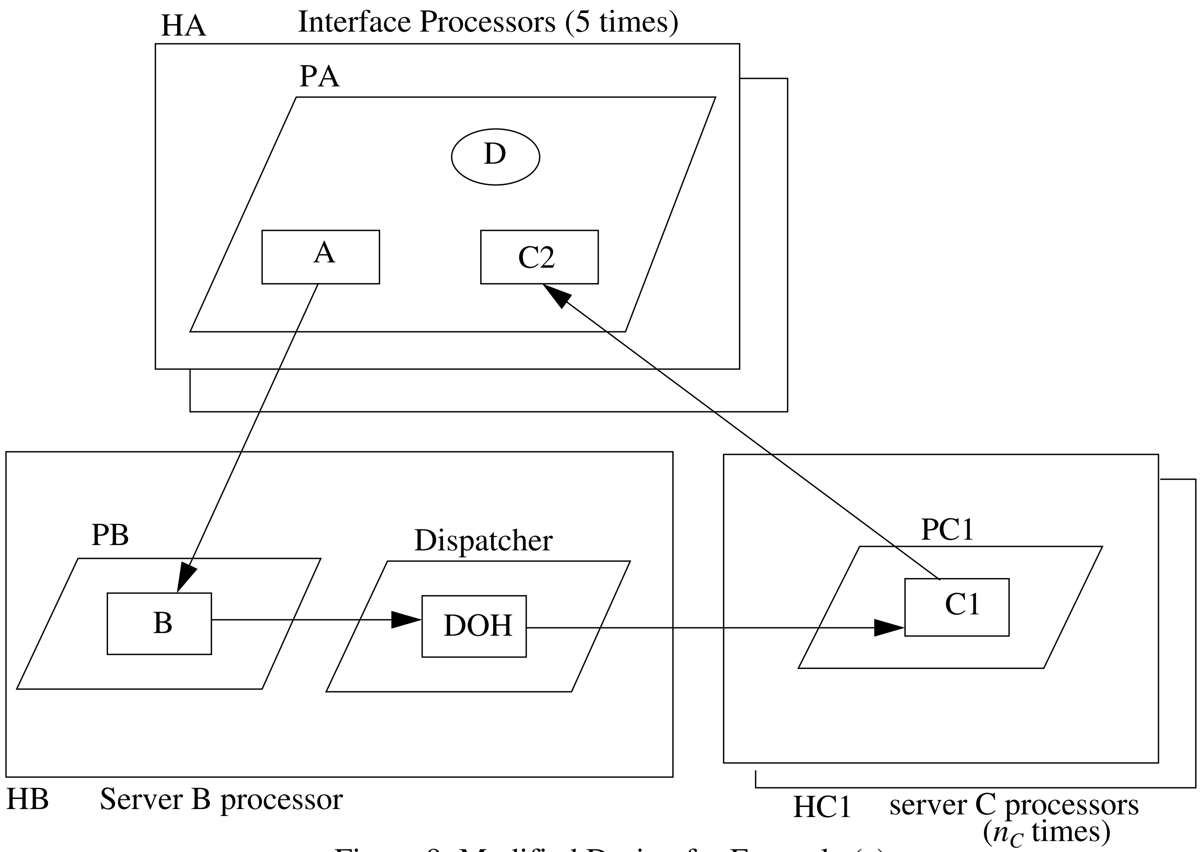


Figure 9: Modified Design for Example (a)

Table 3: Example (a) Analysis by Processor at a Fixed Path Frequency of (5 x 1/3)/sec.

	Processor	Activities	Exec. per activity	demand	capacity	utilization
Original System of Fig 4(d)	HA	A + C2	1.1	1.84	5.	37%
	HB	B	0.2	0.33	1.	33%
	HC1	C1	1.0	1.67	1.	167%
Modified System of Fig. 9	HA	A + C2	1.1	1.84	5	36%
	HB	B + DOH	0.3	0.5	1	50%
	HC1	C1	1.0	2.5	n_C	$(167/n_C)$%

Table 4: Example (a): Path-View Performance Results with n_C Copies of PC

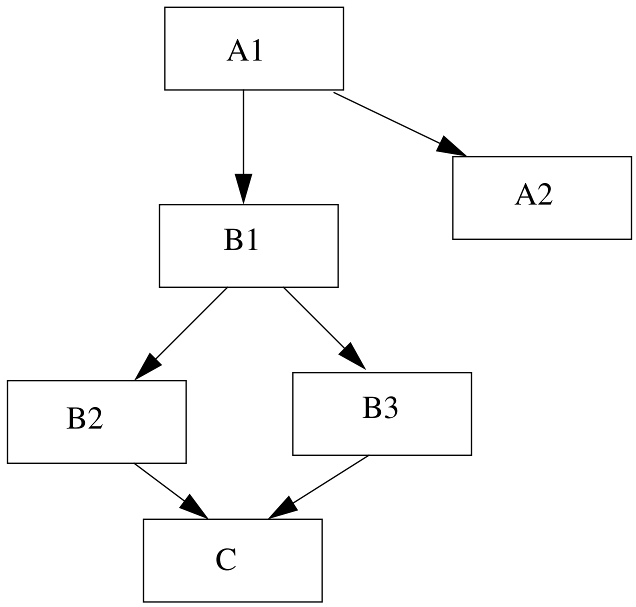
	Activity execution (sec)	delay including contention, once A is begin (sec)			
		n_C=2	n_C=3	n_C=4	n_C=5
A	1.0	1.0	1.0	1.0	1.0
B + DOH	0.3	0.47	0.49	0.49	0.49
C1	1.0	1.41	1.06	1.0	1.0
C2	0.1	0.1	0.1	0.1	0.1
Total time once A is begun	2.4	2.98	2.651	2.59	2.59
Response time (Total with queueing for PA)		*	5.28	4.68	4.56

Example (b): Delay and Path Restructuring: When a path is too long it may also be possible to restructure it to make better use of parallelism. In the design shown in Figure 4(a) the activities ABC are executed strictly in sequence, from the point of view of each interface. Consider a case where B is quite lengthy and C is short, i.e. t_A=1, t_B=1, t_C=0.2, and there is a bottleneck at B. There are five interfaces, each with one interface process and processor running as fast as they can, taking a new input as soon as the previous processing of C for that interface is completed. The throughput, limited by B, is 0.99/sec. The performance results (found from a Markovian analysis, assuming exponentially distributed random activity times) are shown in Table 5, organized as a pivot table for this system. The pivot element is B, identified by fact that the system throughput is mainly limited by the saturation of PB and HB (because they are 100% utilized), and by the fact that the delay at PB is the major part of the total path delay.

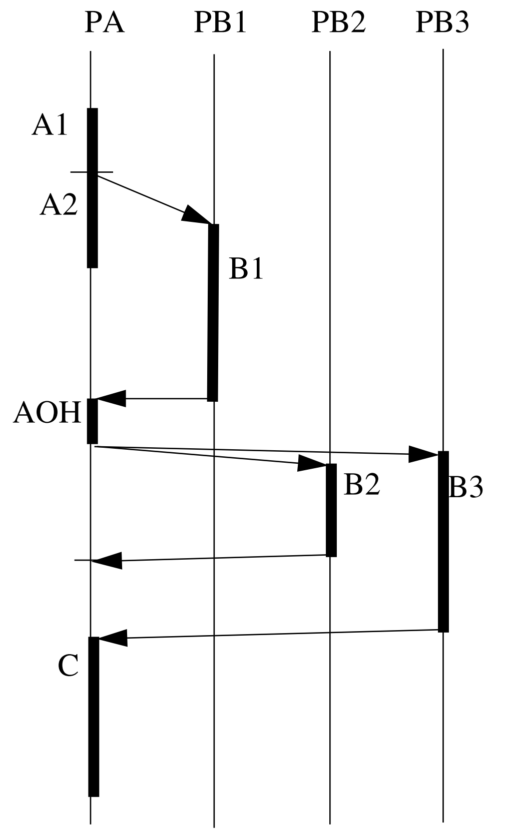
Table 5: Pivot Table for Example (b) before Redesign.

Map View		Resource View			Path View		
Process	Load (sec/resp)	Processor	Multiplicity	Utilization	Activity	Time	Delay
PA	1.2	HA	5	0.24	A	1.0	1.00
PB	**1.0**	**HB**	**1**	**0.99**	**B**	**1.0**	**3.85**
					C	0.2	0.20
					Total		5.05

What can be done to break the bottleneck at B in this case? Suppose that by re-examining the requirements for A and B we have found that they can be modified to allow some parallelism. First, A can be divided into a part A1 that must be completed before sending the request to B, and a part A2 that can be completed after sending the request. Second, B can be divided into three parts, each implemented in a separate process and processor and each handling its own portion of the data-consistency problem that led originally to B being serialized. One part B1 must be executed first, and then B2 and B3 can be run in parallel (while B1 can be run for the next request). Figure 10(a) shows the revised path. The structure is like Figure 4(a), except with three processes PB1, PB2, PB3 in place of PB. Since it was originally stated that PB was a pre-existing server process, this is a drastic redesign. PA now executes activities A1, A2, and C. Figure 10(b) shows a message diagram, in which PA first makes a blocking request to PB1, waits for the response, and then makes two parallel requests to PB2 and PB3. It waits for both responses before executing C. The penalty paid for overhead in handling the smaller units is an extra 0.1 sec for overhead activity AOH in PA and for overhead in each of B1, B2, B3. Thus if B is divided into three equal parts, each part has a time of 0.433 sec. (1/3 + 0.1). The message diagram shows AOH expended in sending requests to PB2 and PB3, although in fact it may be distributed between sending requests and receiving responses. A separate thread of control is set up to complete A2, so it does not interfere with the interactions with PB1, PB2 and PB3.



(a) Revised Path



(b) Message Diagram showing Interaction of PA with Three Server Processes PB1, PB2, PB3.

Figure 10: Modified Design for Example (b)

The performance was modelled as before, for five event streams with one PA per stream, and one copy each of PB1, PB2, PB3, PC. Table 6 shows the path time components of the original system, and the modified system. The “path delay” column is exactly the same as “exec. plus waiting” except for the time to complete B2 and B3 in parallel, where it is the expected value of the maximum of the two delays. The redesigned system is quite successful in decreasing the path time, and obtains a nearly doubled throughput.

Table 6: Example (b): Performance Results with Parallelization: Free-Running

	Activity	exec.	exec. plus waiting for processing	Path delay contribution	Throughput = 5/(path delay sum)
Original System of Fig 4(b)	A	1.0	1.00	1.00	
	B	1.0	3.85	3.85	
	C	0.2	0.20	0.20	
	Sum	2.20	5.05	5.05	0.99
Modified System of Fig. 10	A1	0.5	0.6	0.6	
	A2	0.5	0.5	--	
	B1 + B1OH	0.433	.757	.757	
	AOH	0.1	0.1	0.1	
	B2 and B3 with overhead	0.433 each	.775 each	--	
	B2 and B3 plus sync delay	--	--	1.11	
	C	0.2	0.2	0.2	
	Sum	2.60		2.67	1.87

Compatible Levels of Abstraction

The problem of cross-referencing activities and modules when both models use hierarchical abstraction was mentioned in Section 4. In the examples (a) and (b) of this section, the view descriptions are closely related -- they are at compatible levels of abstraction. Compatible levels of abstraction are the most appropriate for conveying information between views or for pivoting between views. The measure of compatibility is the degree to which events in one view can be matched to points in another. For example certain events in a path correspond to certain interface crossings in the map, and perhaps also to certain scheduling events in the resource view. In many cases the correct levels for compatibility will be clear to the user; in more complex cases some navigational assistance may be necessary for finding compatible levels of abstraction.

6 A Taxonomy of Software Performance Processes

Once the three views are distinguished, the different approaches which may be used for performance engineering become more understandable. The Introduction noted that there are path-dominated and map-dominated approaches to SPE. This section will describe a wide range of approaches by the way each one uses the three views M (map), P (path), R (resource), in an order which begins with the dominating view, defined first, and follows the sequence of use. For example the *path dominated* technique of cite-CHU91 concentrates most of its effort in developing the path model (in their case, a task graph or partial order model), but then shifts to a queueing model to obtain resource-contention delay values which are needed to complete the path delay computation. We will call this a *P-R* (path-resource) technique, because the resource analysis follows the path definition and is slaved to it. Smith has a more complete *P-R* process in cite-CUS90, with a balance between improvement and analysis, and between correctly configuring the resources and describing and improving the paths.

A *map-dominated* process was also mentioned in the Introduction, using either object-oriented design or a simulation based on the design of the software components. If the design can be executed as a simulation, including simulated processors with scheduling and timing, it can provide performance estimates and give an *M-R* approach. Other potential *M-R* approaches are described below.

The two examples in the previous section showed the intertwined use of the views in problem-solving, which seems to be the ideal. However in a larger perspective, reviewing the three examples, it seems that a natural procedure begins in P, goes to M to do checks, determine overhead and reconstruct the path to correspond to the design, then back to P to account for the path reconstruction, and to R for evaluation. This is a *P-M-P-R* approach.

Some further alternatives will now be described. This will suggest new possible ways of addressing performance engineering, and put them in perspective. Ideally it will help us to construct an approach which makes the best use of all the information available in each given situation. Often there are fragments of performance data -- partial workload information, design fragments to be re-used, or processor information -- that cannot be exploited early enough because they cannot be connected together. The three-view framework should have a slot for each piece of information, and machinery for exploiting whatever is known at a given time.

P-M-R: This seems to be a major candidate for a complete performance process, as it extends the *P-R* process (as expounded for example by Smith) to include and exploit map information. It also corresponds to the most commonly described stages of development, beginning with path modelling for the analysis stage. The path definition complements a data-flow model by identifying the sequences of greatest concern, with activities corresponding to processes in the data flow model. The map model is then a simple interpretation of the design of the software components, and identifies boundary-crossing overheads. The resource model corresponds to system integration in the target processing environment.

Just because the views correspond to stages of development does not mean that a view is only constructed once that stage is reached. Instead, the known existence and structure of the views can be used to look ahead to stages that are not yet reached, as indicated in Figure 2(b). In the analysis stage of a *P-M-R* method the path view is filled out as much as possible, including predicted activity times. Then an *extrapolated map model* is made up from

- predetermined map components that are being re-used
- sketched-in subsystem boundaries provided by the designer
- a default module structure inferred from the activity structure.

For example the activities at the highest level of abstraction could be identified with active or passive modules, and the lower level activities with procedures. This could be done in a similar way to the default "transform structure" which is sometimes suggested as a first step towards a structured design cite-Yourdon2. It would not be seriously intended as a design, but could be used for making early performance predictions at Check #1 in Figure 2(b).

Similarly an extrapolated resource model is constructed from

- a high-level description of the processors and interconnections
- a full description of those devices that have been definitely selected.

Often the processor configuration is decided quite early, and full information is available. At this point the first performance prediction can be made. If it is unsatisfactory, the analysis stage can be revised before going on to the design.

During the design stage the map model can track the development of the design, keeping sketched-in or default modules for parts which are not yet designed. At every step the performance model can be re-solved to evaluate the effect of design decisions, to give Check #2. Once testing begins, unit tests can provide actual activity times and the workload times can be recalibrated, for Check #3.

P-M-P-R (as an improvement on *P-M-R*): It was emphasized in the motivation for using the map view that it has an important role in augmenting the path information, to make it conform to the realities of the design. This means going back to the path view before attempting the evaluation, and modifying the *P-M-R* sequence to *P-M-P-R*. The second visit to P inserts overhead activities into the paths for coordination of activities in a single module, and for message passing, context-switching, and procedure calls. It also verifies that nothing is left out in M. The examples in Section 5 follow this sequence.

M-P-R: This class of processes begins with decisions on the software components. One way this may happen is in object-oriented analysis as described by Jacobson et al cite-jacobson92 or by Shlaer and Mellor cite-ShlaerMellor92 which identifies potential software objects and their data structures and methods. As these are future software components, this creates a map description in which *objects* are modules, and *methods* execute activities and also access the methods (and activities) of other objects. A performance engineering process for object-oriented design would next construct a path model as in the interaction diagrams of Jacobson's OOSE method cite-jacobson92, tracing the important paths through the methods, and do the usual path operations to determine frequencies of activity execution, etc., for workload analysis. It would also assemble an extrapolated resource model, as in the *P-M-R* process, and use it to predict performance figures for the design. Design tuning would then imply modifying the objects in M, reconstructing the paths as appropriate, and re-solving for performance.

P-R-M-R**:** This begins similarly to *P-M-R* but then analyzes the performance potential of the physical architecture using the resource view, before creating the map entities. The design is thus matched to the resources available. The activities are allocated first to the processors to see how the physical resources should be employed to achieve the paths. This could in principle be done as an optimization calculation, provided all the constraints which the design must satisfy are expressed in the optimization; however it is a difficult and unsolved problem. The final resource analysis is a verification. A desirable final extension adds the path reconstruction for overhead activities and control affects, after the M stage, giving a *P-R-M-P-R* process.

M-R for client-server systems: *M-R* approaches are suitable for designing systems which use pre-existing components and which have very simple stereotyped interactions. This is true for example of client-server systems modelled by the SP process of Vetland, Hughes and Solvberg cite-SP93 or by Rendezvous Networks cite-SRVN, where a relatively simple architectural description is sufficient to generate the set of all paths, and to give performance predictions.

M-R-P-R**:** An extension of *M-R* in which the generic structure-based models are used for the main design effort, but heavily-used paths are identified and checked before finishing the design.

There may be further distinct processes. However, it is not very helpful to identify processes which repeat some steps in a loop, as being different. Also, it seems reasonable that R should always appear last as a final performance estimate. Finally, it does not seem to be reasonable to put the resource view first, as there is no workload to analyze until at least one other view is described. Then the above list appears to cover the full range of performance engineering processes.

One potential weakness of all the above methods is the use of predicted values for activity times, which may not be accurate. Once the path model has identified the most frequently-executed activities, the risk of wrong estimation of their execution times can be reduced by prototyping the individual activities. Re-use of existing modules makes it possible to also re-use their activity data. However for new portions of a system some risk associated with using predicted values must be tolerated.

7 Conclusions

Performance engineering is primarily a problem of information management. A good process must rest on a high-level data model which connects together the various kinds of data relating to performance and design. This paper has presented a framework for software performance engineering with a data model which is more complete than those previously used in the “Core Model” of Section 3. Section 4 shows how some well-known design and modelling viewpoints (each with its own well-worked-out data definitions) relate to the Core and to the framework of this paper. Some short examples demonstrate how the framework can be used on performance-related design problems involving concurrency, and the paper concludes with a taxonomy of software performance *processes* based on the framework. Using the reasoning supporting this taxonomy, a customized process can be chosen to match the needs of a given software project.

Acknowledgements

This work was supported by the Telecommunications Research Institute of Ontario, and by the Strategic Grants Program of the Natural Sciences and Engineering Research Council of Canada. Many people commented on early drafts of this material; I am particularly grateful to Gerald Karam, Maged Beshai, John Neilson, Shikharesh Majumdar, Dorina Petriu and Curtis Hrischuk.

Bibliography