Performance by Unified Model Analysis (PUMA)

Murray Woodside, Dorina C. Petriu,
Dorin B. Petriu, Hui Shen, Toqeer Israr
Dept. of Systems and Computer Engineering,
Carleton University, Ottawa, Canada
{cmw | petriu | dorin | hshen | tisrar} @sce.carleton.ca

Jose Merseguer
Dep. de Informatica e Ingenieria de Sistemas,
Universidad de Zaragoza, Zaragoza, Spain
jmerse@unizar.es

ABSTRACT

Evaluation of non-functional properties of a design (such as performance, dependability, security, etc.) can be enabled by design annotations specific to the property to be evaluated. Performance properties, for instance, can be annotated on UML designs by using the "UML Profile for Schedulability, Performance and Time (SPT)". However the communication between the design description in UML and the tools used for non-functional properties evaluation requires support, particularly for performance where there are many alternative performance analysis tools that might be applied. This paper describes a tool architecture called PUMA, which provides a unified interface between different kinds of design information and different kinds of performance models, for example Markov models, stochastic Petri nets and process algebras, queues and layered queues.

The paper concentrates on the creation of performance models. The unified interface of PUMA is centered on an intermediate model called Core Scenario Model (CSM), which is extracted from the annotated design model. Experience shows that CSM is also necessary for cleaning and auditing the design information, and providing default interpretations in case it is incomplete, before creating a performance model.

Keywords

Software performance engineering, performance models, UML, scenarios, model building.

1. INTRODUCTION

Considerable emphasis has been placed on developing an ability to evaluate software and system designs for non-functional properties such as performance, reliability, and security. One approach to enabling this evaluation is to attach suitable additional information as annotations to the design. This has been addressed for performance and schedulability in the standard "UML Profile for Schedulability, Performance and Time" (SPT) [13]. The SPT profile defines stereotypes and tagged values that can be attached to design model elements, particularly in the behaviour and deployment specifications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSP'05, July 11–15, 2005, Palma de Mallorca, Spain.
Copyright 2005 ACM 1-59593-087-6/05/0007 …$5.00.

Translations from UML into different kinds of performance models have been described, for example:

- into queueing models, by Smith [19]
- into layered queueing models [14][16]
- into stochastic Petri nets [3][6][11]
- into stochastic process algebra models [4]
- directly into simulation models [2]

The model translation can be somewhat intricate, and the approaches these authors have taken to interpreting the UML are affected by the target performance semantics. Further, each contribution addresses one kind of UML diagrams, such as sequence diagrams, activity diagrams and state machines, which do not express behaviour in the same way.

A software group would prefer to have access to several kinds of performance model and tools, from their preferred software design tools. Also, design notations provide many ways to model a system (e.g, within UML, scenarios can be described either by interaction or by activity diagrams) and different versions of UML have different metamodels and semantics. Thus we have a kind of *N-by-M* problem to translate *N* design notation types into *M* performance model types.

N-by-M problems are best addressed by a common intermediate format, such as the Core Scenario Model (CSM) described below. It captures the essence of performance specification and estimation as expressed in the SPT Profile, and strips away the design detail which is irrelevant to that analysis. It is suited to the production of performance models of several kinds, as demonstrated here by layered and regular queuing networks, and stochastic Petri nets. It is equally suited to different UML diagrams, as it is derived directly from the SPT profile. It can be used with non-UML software specifications as well, such as Use Case Maps. Other intermediate performance models have been described, such as Execution Graphs (Smith [18] and Cortellessa [5]). CSM is proposed as a standard for these ideas, compatible with the standard UML SPT profile.

CSM is not only a common format, but it is much closer to all the target performance models, than the XMI encoding of UML. The information in CSM is filtered and verified; only the performance-related model elements are included.

1.1 PUMA architecture

The PUMA architecture is a framework into which different kinds of software design tools (first and foremost, UML tools) can be plugged as sources, and different kinds of performance tools can be plugged as targets. The complete tool architecture is indicated in Figure 1, including not just evaluation, but also exploration of the performance properties of the design, and feedback to the design domain.

For UML tools, the input to the CSM translator is the XML format (XMI). Other specification languages must either generate XMI, or (as in the case of Use Case Maps) have a customized translator. The CSM is expressed in CSML, with its own meta-model [15], and from CSML each performance model type has its own translator.

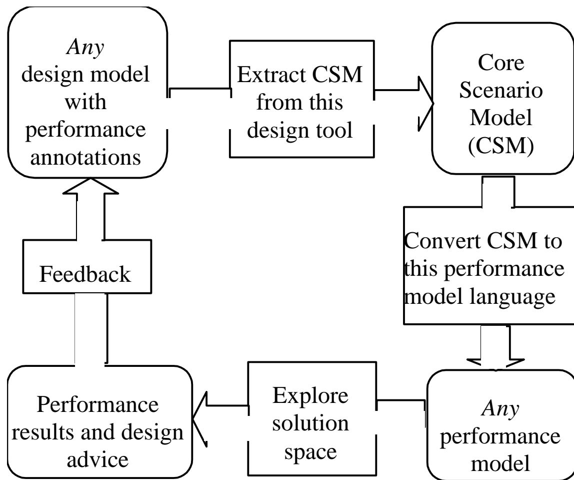

Figure 1 The PUMA Architecture

We will describe transformations into CSM from UML1.4 (Activity and Deployment diagrams), fromUML2 (Interaction and Deployment diagrams), and from CSM into Layered Queueing Networks (LQN), Petri Nets (PN), and Queueing Networks (QN).

1.2 Running Example

This paper will present the PUMA tools through a running example, which has been used earlier to explain the use of the SPT profile [17]. It is a Building Security System (BSS), with a part that stores video frames from surveillance cameras, and a part that manages keyed-number access controls to doors in the building. The discussion here will only address the video scenario, because of space limitations.

The Profile is based on scenarios (that is, realizations of Use Cases given as behaviour diagrams), and deployment. Figure 2 shows the deployment of the software on two processors on a LAN, with SPT stereotypes `<<PAhost>>` to indicate that they are host processors to the software components shown for each. It is expected that this diagram indicates how are deployed those software components that are participating in the scenarios considered for the generation of the performance model.

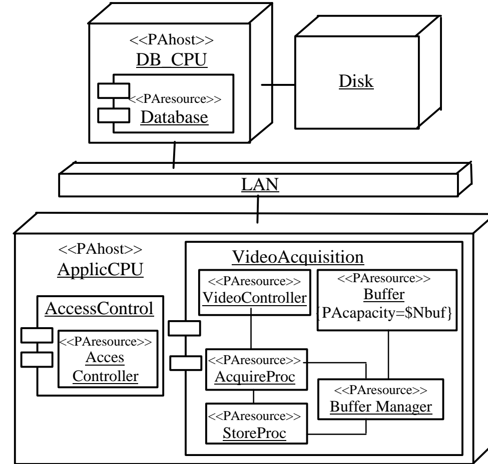

Figure 2 Deployment and software components in the Building Security System (BSS)

A buffer pool indicated as `Buffer` in the `VideoAcqui-sition` component is a storage resource, stereotyped as `<<PAresource>>;` it is referenced in the video scenario. It has a multiplicity parameter `$NBuf`, which is the number of buffers in the pool.

Activity Diagram

Figure 3 shows a UML1.4 Activity Diagram for the behaviour of the video acquisition scenario. There are `$N` cameras to be polled in a cycle, giving frames which are buffered and then stored in the Database. Steps in the scenario represent the workload of operations, and may be stereotyped on the message which initiates the operation, or on the execution occurence or activity that executes it. In Fig. 3 `<<PAstep>>` stereotypes are attached to activities by notes.

The first Step, `cycleInit`, is executed only once per cycle and has attached to it two stereotypes; one for its own resource demands, such as CPU time, and another for the *Workload* of the whole scenario. Here the workload consists of a single initiator or token, which repeatedly (without any inserted delay) triggers camera scanning cycles. The workload stereotype `<<PAclosedLoad>>` also defines the end-to-end performance requirements as an interval constraint of 1 second between successive repetitions of the cycle, in 95% of cycles. Finally it also defines a variable name `$Cycle` for the actual 95% delay, which will be obtained from the analysis of the performance model. The Step `cycleInit` is followed by a loop with `$N` repetitions, which is represented at the top of Fig 3 by a composite activity `procOneImage` (for one cycle), with a repetition count of `$N`. The refinement of `procOneImage` is given in the lower part of the figure.

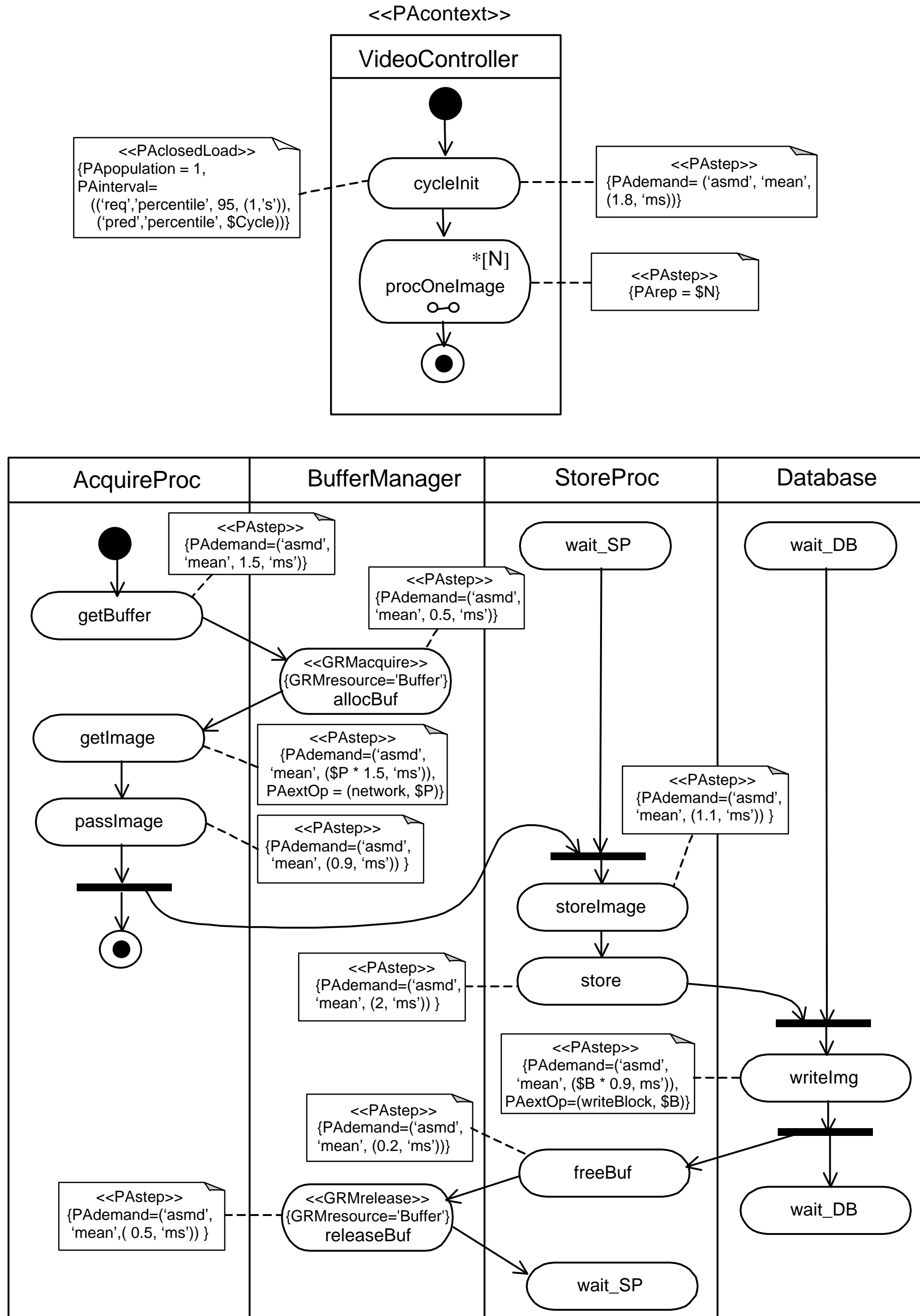

Figure 3 UML1.4 Activity Diagram for the Acquire/Store Video Scenario for the building security system

Fig. 3 uses swimlanes (the vertical strips) to represent behaviour of concurrent software components. Each swimlane is associated with a component name; this is a UML 2.0 feature, but we also used it with UML 1.4 as a PUMA convention. If the component corresponding to a swimlane is stereotyped `<<PAresource>>`, this indicates that the component runs within a process that must be obtained before execution commences (i.e., request messages are queued before being accepted and executed). If such a component has a `{PACapacity}` parameter, it indicates a multi-threaded process. The sequence of activities (which become CSM *Steps)* is clearly established from the connectors between activities, including forking and joining at the horizontal bars. The `allocBuffer` step by the `BufferManager` process involves a second stereotype `<<GRMAcquire>>`, because this step acquires a buffer resource able to contain one video frame. If no buffer is available (indicating buffer starvation which may limit performance), then the requesting thread is blocked. The

use of the `<<GRMacquire>>` stereotype (and its counterpart `<<GRMrelease>>`) for logical resources such as a buffer pool is suggested in the SPT Profile but not fully defined, and the use of a parameter to identify the resource (by name) is our own extension of the Profile. This extension appears to be necessary for resources other than processes and nodes, which are implicit.

The terms "active" and "passive" are used differently in UML and in the SPT Profile (and consequently CSM). The SPT Profile distinguishes active resources that can initiate events (typically hardware, such as processors) and passive resources that only respond to events (e.g., processes and buffers). UML in general recognizes active and passive software components, where a process is active, while a regular object is passive. Thus, in SPT and CSM processes and buffers are both passive resources, while in UML a process is an active component and a buffer is a passive component.

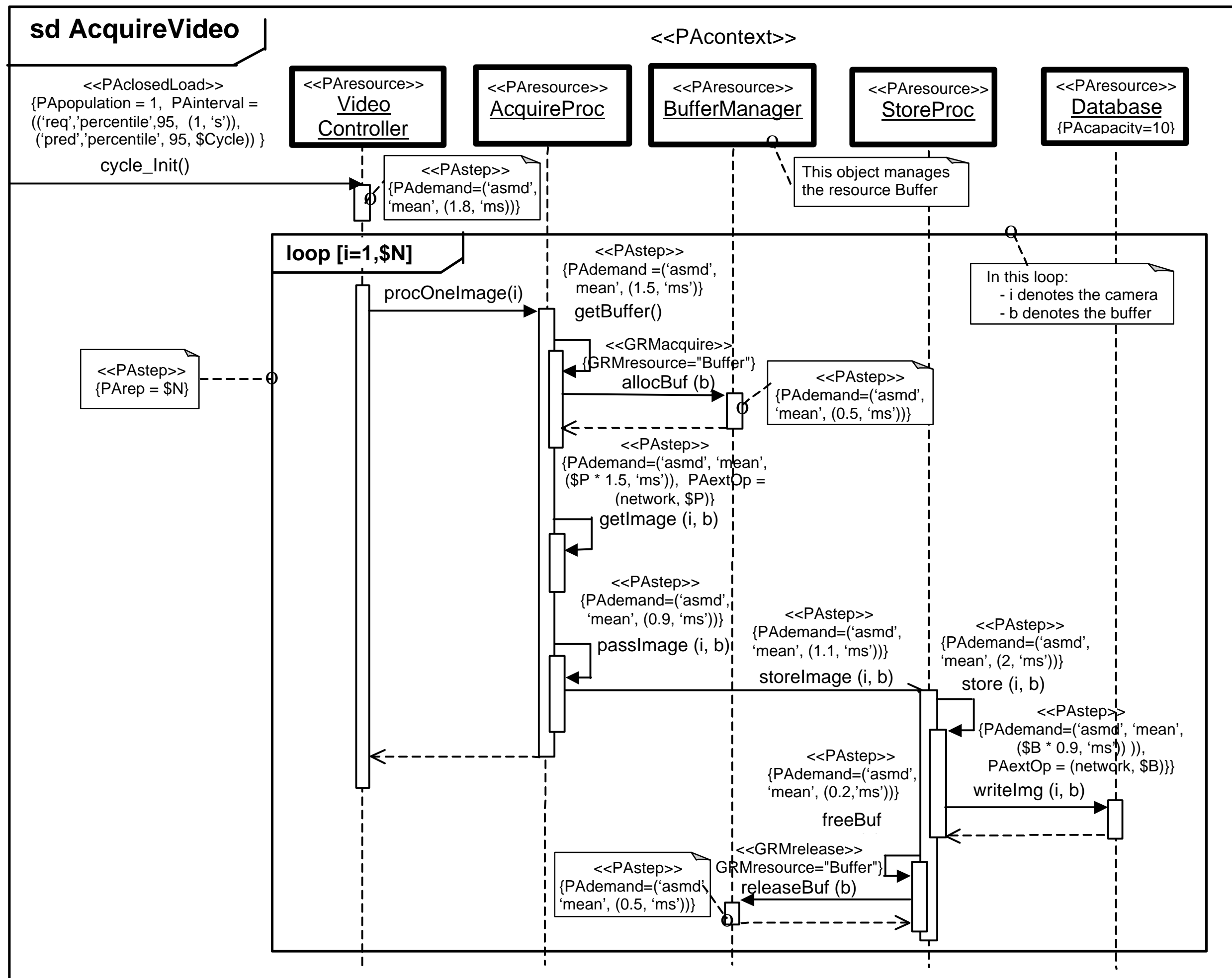

Figure 4. A UML2 Interaction Diagram for the Acquire/Store Video scenario for the Building Security System from [15]

Figure 4 shows a UML 2.0 Interaction Diagram for the same scenario. For some steps the use of UML notes to attach the stereotypes is illustrated here. Notes may be useful with some tools which do not produce XMI output for stereotypes.

The scenario definitions include a pipeline effect in which `AcquireProc` passes the buffer to `StoreProc`, which handles storage in the database, while `AcquireProc` returns to poll the next camera in the cycle.

2. THE CORE SCENARIO MODEL

The Core Scenario Model was presented in [14] and is based closely on the domain model of the SPT profile. A *Scenario* is a sequence of *Steps*, linked by *Connectors* that include sequence, branch/merge (OR fork/join), fork/join (AND fork/join), and *Start* and *End* points. A *Scenario* uses *Resources,* which may be Active (including host processors which execute steps) or Passive (including logical resources, which are explicitly acquired and released by special *ResAcquire* and *ResRelease*

steps). Resources are defined with a discipline (such as FIFO). Start points are associated with a *Workload* that defines arrivals and customers, and may be open or closed. Steps are executed by *Components*, (software components) which are passive resources in CSM. A *Step* may include a *sub-Scenario* as a refinement, and a loop is described by a *Step* with a repetition count and a refinement for the loop body.

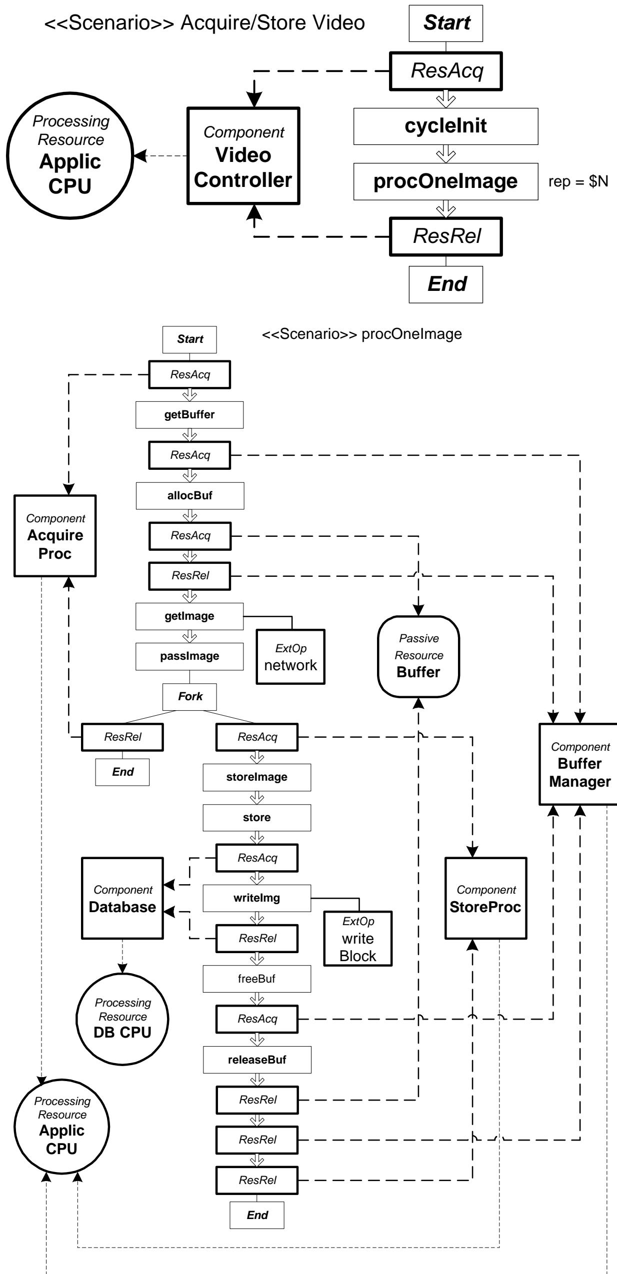

Figure 5 Core Scenario Model for the Video Scenario.

A *Step* is a sequential operation using a host processor that is identified by the deployment of its *Component*. It may also include *External Operations*, which are not modeled within the UML design, but are known to be required (e.g. file and database operations). They are identified by name, frequency (mean number of operations during the *Step*) and delay.

Figure 5 shows an example CSM representation of the UML Acquire-video scenario. The figure contains in fact two CSM scenarios: the first one describes the entire video acquisition operation as a step, which is repeated for each camera, and the second describes the sub-scenario for that step. The resources are represented with bold outlines: circles for processing resources, squares for components and rounded squares for passive resources. The steps are represented as rectangles, and the dependencies between steps and resources as dotted arrows. A special hollow arrow represents a *Sequence* path connection and its associations with the predecessor and successor steps.

2.1 From UML1.4 to CSM

UML tools are supposed to export the annotated model to an XML file according to the standard UML-to-XML interchange format XMI, which PUMA translators take as input. The strategy in translating XMI to CSM is to find the diagrams that are to be considered, by searching for SPT Profile stereotypes, and then to generate structural CSM elements (*Resources* and *Components*) from the deployment diagram, and behavioural elements (*Scenarios*, *Steps* and *Path-Connections*) from the behaviour diagrams. This section describes briefly the translation from UML1.4 deployment and activity diagrams to CSM (the UML 1.4 interaction diagrams are too restricted, so are not described here). The next section will discuss the translation of UML 2.0 sequence diagrams, which have been enhanced considerably with respect to UML 1.4.

The translation algorithm begins with the deployment diagram (for simplicity, we assume that there is only one such diagram in the UML model). A UML *Node* translates into a CSM *Resource* (specifically into a *ProcessingResource* if stereotyped as <<PAhost>>). A UML *Component* translates into a CSM *Component* if it is stereotyped with <<PAresource>>, and into a CSM *PassiveComponent* otherwise. An object with stereotype <<GRMResource>> translates to a *PassiveResource*.

The translation continues with activity diagrams stereotyped as `<<PAcontext>>`. For each one, the *Initial* pseudostate, which is translated to a CSM *Start* PathConnection and a *ResourceAcquire* step for acquiring the component for the respective swimlane. The scenario workload information in a *Workload* stereotype is translated to a CSM *Workload* element attached to the *Start* PathConnection. (Note that in the SPT Profile, the scenario workload information is associated by convention with the first step of a scenario, not with its *Initial PseudoState*). The translation follows the sequence of the scenario from start to finish, identifying the *Steps* and *PathConnections* (representing sequence, branch/merge, and fork/join) from the context of the diagram. Each simple activity represented by a UML *ActionState* is translated to a CSM *Step*, whereas a UML *CompositeState* gives a CSM *Step* with a nested *Scenario*.

Each swimlane is associated with a *Component* through its name. A UML *Transition* that crosses the swimlane boundary (named here a "cross-transition") represents a message or signal between the corresponding components that implies releasing the sender (which is a *Component* and also a *Resource*) and

acquiring the receiver. Therefore, a cross-transition translates to a CSM *ResourceRelease* step, a *Sequence PathConnection* and a *ResourceAcquire* step.

2.2 From UML2.0 Interaction Diagrams to CSM

UML 2.0 interaction diagrams have better capabilities to model scenarios than previous UML sequence diagrams. It is possible to express selection among alternate branches, parallel execution, loops, etc., by using so-called *fragments*. A *combined fragment* encapsulates a portion of a sequence diagram surrounded by a frame, and contains one or more operand regions tiled vertically and separated by horizontal dashed lines. An operator shown in the upper-left corner of the frame prescribes how the operand regions of the combined fragment are handled. For instance, the operators `opt` and `alt` are used for branch selection, `par` for parallel execution and `loop` for repetition. Another new feature allows for hierarchical decomposition of a scenario step into a more detailed sub-scenario. This is done by using an *interaction occurrence*, a fragment labeled with the operator `ref`, which refers to another interaction shown in a separate sequence diagram.

The SPT Profile, which was defined for UML1.4 and has not been yet upgraded for UML 2.0, needs to be extended to allow for stereotyping of fragment operand regions and interaction occurrences with `<<PAstep>>`. For example, the `loop` fragment shown in Figure 2 is stereotyped as a step with a `PArep` attribute giving the number of repetitions.

The translation algorithm from UML 2.0 to CSM begins with the deployment diagram, similarly with the translation from UML1.4. A UML *Node* is converted into a CSM *ProcessingResource* if stereotyped as `<<PAhost>>`, and into a CSM *Resource* otherwise. A UML *Artifact* (a deployable entity that “manifests” a component) is converted into a CSM *PassiveResource* if its stereotype is `<<PAresource>>`, and into a CSM passive *Component* otherwise.

The translation continues with the scenarios described by sequence diagrams stereotyped with `<<PAcontext>>`. For each scenario, a CSM *Start PathConnection* is generated first, and the workload information is attached to it. Each *Lifeline* from a sequence diagram describes the behaviour of a UML instance (be it active or passive) and corresponds in turn to a CSM *Component*. We assume that the artifacts for all active UML instances are shown on the deployment diagram, so their corresponding CSM *Components* were already generated. However, it is possible that the sequence diagram contains lifelines for passive objects not shown in the deployment diagram. In such a case, the corresponding CSM *Passive Component* is generated, and its host is inferred to be the same as that of the active component in whose context it executes.

The translation follows the message flow of the scenario, generating the corresponding *Steps* and *PathConnections*. A simple *Step* corresponds to a UML *Execution Occurrence*, which is the effect of receiving a message. Complex CSM *Steps* with a nested scenario correspond to operand regions of UML *Combined Fragments* and *Interaction Occurrences*. A synchronous message will generate a CSM *Sequence PathConnection* between the step sending the message and the step executed as an effect. An asynchronous message spawns a parallel thread, and thus will generate a *Fork PathConnection* with two outgoing paths: one follows the sender's activity, and the other follows the path of the message. The two paths may rejoin later through a *Join PathConnection*. Fork/join of parallel paths may be also generated by a `par` *Combined Fragment*. Conditional execution of alternate paths is generated by `alt` and `opt` *Combined Fragments*.

Because some CSM *Components* are also *Resources*, additional resource acquire and release steps may be necessary. More specifically, if the sender of a message is stereotyped as `<<PAresource>>` a *ResourceRelease* step is generated, and if the receiver is stereotyped as `<<PAresource>>` a *ResourceAcquire* step is generated.

The translation of UML 1.4 sequence diagrams to scenarios is a simple case of the translation discussed above for UML 2.0. In UML1.4 there are no combined fragments for expressing selections, repetitions and parallel executions, nor interaction occurrences for expressing further step refinements.

3. CHALLENGES IN CSM EXTRACTION

A UML model contains multiple system views described by different types of diagrams. However, only some of the information contained in the design specification is relevant to performance evaluation. Therefore, one of the challenges for CSM extraction is to identify what is necessary for generating a performance model and to filter out irrelevant information.

Another problem is that UML design models may be incomplete or inconsistent, especially in the early stages of software development. Software design tools are required to support incomplete specifications that designers use for documentation and discussion of the evolution of the concepts in a system. However, the generated CSM should be complete and well-formed before proceeding to the next step, the generation of performance models. Thus, another challenge for CSM extraction is dealing with the incompleteness/ inconsistency of both the UML model and of its performance annotations.

3.1 Integration across diagrams

The information needed to define a performance model is spread across multiple diagrams with differences in their semantics. The first challenge is to find the diagrams that are relevant, especially the behaviour diagrams defining scenarios, which are stereotyped `<<PAcontext>>`. Within these diagrams run-time instances (components) are referenced directly (e.g., by lifelines in a sequence/interaction diagram), or indirectly (e.g., as through the labeling of swimlanes in an activity diagram). In this work we shall assume that a swimlane contains activities executed by a single run-time component whose name is referenced by the swimlane label, although this interpretation is not necessary in UML.

Deployment of active components (processes) is determined from the deployment diagram (we shall assume there is just one such diagram for simplicity). Passive components are taken to be deployed on the same host as the active component in whose context they execute, as shown by calling patterns in the behaviour diagram.

The active or passive nature of components must be determined. This may be found in different ways: (1) as the attribute "isActive" of the class of the instance explicitly set in the UML model; (2) from stereotyping of the instance as <<PAresource>> in a interaction or deployment diagram; or (3) from associating the instance with a swimlane in the activity diagram. If this cannot be determined, the user can be queried, or a default interpretation as an active component deployed on a virtual processor is taken.

Multiple diagrams for a system may require reconciliation. A stereotype may be applied to an object in one but not another. In some cases, one diagram may fill in detail from another (as in a compound activity in UML1.4, or an interaction fragment in UML2) and the scenario traversal must track from one to another and back.

The value of CSM is in filtering out the mass of UML definitions, drawing the performance information together, and permitting tests on the CSM for completeness and consistency of the performance-related attributes. These are a combination of the performance-stereotyped information and the structure and attributes of the UML model.

Also, since UML tools differ, CSM provides a boundary to this concern and to UML-tool-specific interpretation. Tools may differ in how XMI is produced (or produced for only some diagrams), or in how UML features are supported. For instance, the optional representation of a branch in UML1.4 by diverging message paths is supported in some tools and not others. UML2 provides a clearer expression for this with its `alt` construct.

3.2 Interpretation of the Design model(s)

We assume that the user indicates what scenarios are to be considered for CSM extraction by stereotyping them with `<<PAcontext>>`. If a behaviour diagram describes a scenario, its steps can be inferred by following the thread of messages and execution occurrences or activities, even if they are not stereotyped. This could be a convenience to designers. The way of dealing with missing step annotations is discussed in the following sections.

3.3 Completion of the CSM to prepare for analysis

Ideally, all information required to generate the performance model should be completely specified in the design model and its annotations. Parameters such as the CPU demands of Steps, branching probabilities, loop counts, arrival rates and user populations would then appear in the performance model and drive its solutions.

On the other hand, experience with performance tools shows the usefulness of providing default values for parameters at the time objects are created (e.g. in the GreatSPN Petri net tool [22] or in the UCM Navigator [14]). Modelers may forget to define some values, or may not know them at first; the default values permit semantic checks on the performance solution. In some cases, default values may even be acceptable in the absence of experimental estimates (e.g., equal probabilities for branching).

Similarly with missing stereotypes, once an activity diagram is stereotyped as `<<PAcontext>>`, its swimlanes should all reference run-time components and its activities should all be steps. If a swimlane or a lifeline does not reference an identifiable component, by default a component can be created for it on a virtual processor.

Deployment information may be missing in the UML model, since deployment is secondary to functional design of software objects. An approach to deal with such a case, presented in [14], supposed an infinite pool of processors as a default, and deployed the software objects on it. In effect, their CPU demands become pure delays on these processors, and processor contention disappears in the model. Of course this may make it more difficult to identify active and passive components; our solution is to make components active by default, and to inform the user. In fact all defaults and interpretations used should be reported by the tool that creates the CSM.

The SPT Profile allows for one kind of explicit incompleteness. If the system uses resources which are not fully described in the design, it identifies the usage as an “external operation” stereotyped `<<PAextOp>>`. For example, in a given UML model the disk and networks may not be modeled in detail, yet the software will require disk and network operations. This is an explicit hook for an external performance submodel, and requires that the model-building process should interpret the name of the external operation correctly (possibly with user help). The CSM simply captures these and passes them on, with a name and a count for the operations invoked by a Step.

3.4 Verification of the information provided

As already mentioned, a design model may be incomplete or inconsistent. Many tools have limited or no facilities for checking completeness and consistency of a design specification. Performance modeling on the other hand makes some demands. Model construction requires that scenarios are continuously connected, and that all resources acquired should be released. Solution environments also require models to satisfy certain properties, for example to be deadlock free. An important role of the CSM is to support verification, and possibly a finalization step to give a satisfactory CSM.

Liveness or termination, and release of resources, are easy to decide for a CSM. It has much simpler execution patterns than a general behaviour specification, because it describes a single response to a single type of input event (although the type of event may be defined to include alternative subtypes, and thus be rather general). The only way to loop back or jump is to loop within a confined scope (the refinement of a composite step); otherwise the execution path never goes back. Thus, termination is guaranteed, for a correctly connected set of Steps, and the constraints for connecting Steps correctly are simple and few. Resource release can be determined by a traversal of the scenario, such as is described for LQN model-building below.

Multiple scenarios may execute concurrently. We assume that they are analyzed separately to give separate CSMs, and then a model can be created to represent their interaction; they interact only through resources (which include logical resources such as semaphores or locks). Interaction anomalies, such as deadlocks or livelocks, could be carried out by constructing and analyzing a transition system for a set of CSMs. It is possible to do this by generating a state-transition Petri net as described in Section 5, and to carry out the analysis in the Petri net domain with existing tools.

If finalization of the CSM requires queries to a software designer they refer to the familiar design document, rather than to the unfamiliar performance modeling environment. Thus CSM must maintain traceable relations of its objects to the UML diagrams. A degree of traceability is maintained by retaining the names of UML design elements in the CSM where possible. If the UML environment has globally unique identifiers for UML elements, they can be retained in the optional "traceability_ref" attribute of every CSM element [15].

4. FROM CSM TO LQN

For each target performance modeling tool, a separate translator must be created. Within a single model type such as Petri Nets, it would be ideal to target a standard language for model interchange like the proposed PNML standard [10], or the proposed PMIL for queueing networks [20]. Here, we used the input languages of each target performance tool.

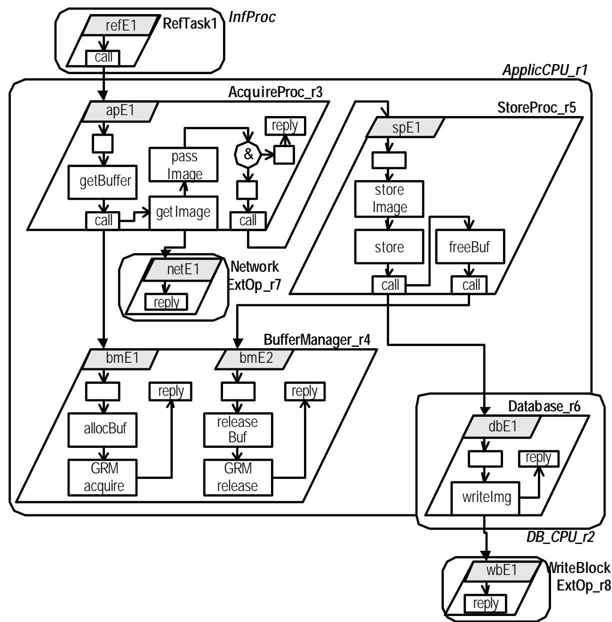

Figure 6. LQN model for the processOneImage scenario

Three model types are considered here, Layered Queueing Networks (LQNs), Petri Nets, and Queueing Networks (QNs).

An algorithm to generate a LQN model from a CSM has been based on the algorithm successfully used to generate models from Use Case Maps, in [14]. Use Case Maps have scenario semantics which are similar to the CSM, as far as workloads, sequential scenario structure and resource acquisition/release are concerned. The first phase of the algorithm generates the LQN resources by examining the CSM resources. The algorithm generates an LQN *Processor* for each CSM *ProcessingResource* and an LQN *Task* for each CSM *Component*.

The second phase of the algorithm traverses the scenario in order to determine the sequencing of the CSM *Steps* and to discover the calling interactions between *Components*. The traversal generates an LQN *Activity* for each CSM *Step* it encounters. Whenever a calling interaction between two *Components* is detected, an *Activity* is created in the *Task* corresponding to the caller *Component* and an LQN *Entry* is created in the *Task* corresponding to the called *Component*. This *Entry* serves the request and its workload is defined by the ensuing *Activities* generated from the *Steps* encountered in the new *Component*. The type of calls is detected by their context, that is a message which returns to a *Component* that previously sent a request is considered to be a reply to a synchronous call. Any calls that have not generated replies by the time the end of the scenario is reached are considered to be asynchronous. As in [14], the algorithm creates a stack of unresolved call messages and removes them as replies are detected (other interaction patterns can also be identified; the reader is referred to [14]). This call stack is duplicated at *Branch* and *Fork* points, so that each ensuing subpath maintains its own message history. The separate call stacks are merged at *Merge* and *Join* points once each incoming subpath has been traversed.

The in-progress version of the algorithm used here still has some limitations. It handles branching and forking but does not yet handle merging and joining of the flow. This is sufficient for the example system, whose LQN model is shown in Figure 6. The logic for creating models with passive resources with finite capacity is not yet implemented, so the Buffer capacity $NBuf was taken as infinite, and the Buffer Pool resource does not appear in the LQN.

An *External Operation* by a CSM *Step* is represented by a LQN request from the corresponding *Activity* to a special *Task* generated to represent it. An option, depending on the tool environment, is to have a library of submodels, which can be included and connected to the generated model through these requests. A CSM *ClosedWorkload* is transformed into parameters for a load-generating *Reference Task,* while a CSM *OpenWorkload* into a stream of requests.

The LQN model is represented in an XML syntax called LQML, for input to the LQN editor, solver, and simulator.

4.1 Multiple Scenarios

One CSM can contain scenarios for several responses, gathered from separate behaviour diagrams. If the scenarios are saved in the same CSM file then the resources are generated once and each scenario is traversed in turn. However if the scenarios are saved in different CSM files then the resources are generated separately for each scenario traversal and separate LQN submodels result. These submodels can later be joined into a single model; the tasks and entries which are common across the different submodels can be automatically merged, based partly on their names and partly on behavioral characteristics.

4.2 Model Exploitation

Only limited information to govern the use of the model is defined in the SPT Profile. An additional interface is needed to define ranges of parameters for sensitivity and scalability evaluation, for instance. We are in process of defining such an interface. If it can be defined at the CSM level it can be used with all the performance tools. However, there are tool-specific issues involved in changing parameter values and re-running a

model in an efficient way. This issue is not resolved, and it is not addressed here for other model types.

5. FROM CSM TO PETRI NETS

A relatively straightforward translation to Generalized Stochastic Petri Nets (GSPNs) (described for instance in [1]) can be implemented using Labeled GSPNs (LGSPNs, described in [3][11][12]). Fragments of a CSM model have direct representations as fragments of LGSPN, with labels that direct the composition of the fragments into a full model. Using the compositional properties of LGSPN, the fragments are composed through several stages until the LGSPN representing the whole scenario is generated. For each class in the CSM metamodel there is a LGSPN pattern, parameterized by the attributes of the CSM class. For example for a *Step*, the basic pattern is a place and a transition as shown in Figure 7(a), where the delay of the transition is defined as the demand attribute of the Step. If the *Step* also requires a *host resource*, the resource must also be acquired, as described below. When the *Step* has a probability less than one, then the translation is given in Figure 7(c) if the Step is preceded by a Branch or in Figure 7(b) if it is not. A Step with a repetition (`repCount`) attribute is shown in Figure 7(d), with the probabilities of transitions t2 and t3 adjusted to give the correct mean count ($\pi 1 = 1/(1 + \text{repCount})$, $\pi 2 = 1 - \pi 1$).

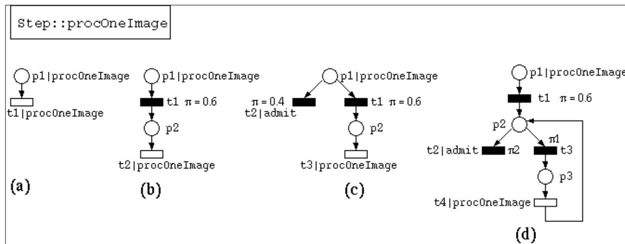

Figure 7 LGSPN patterns for a Step

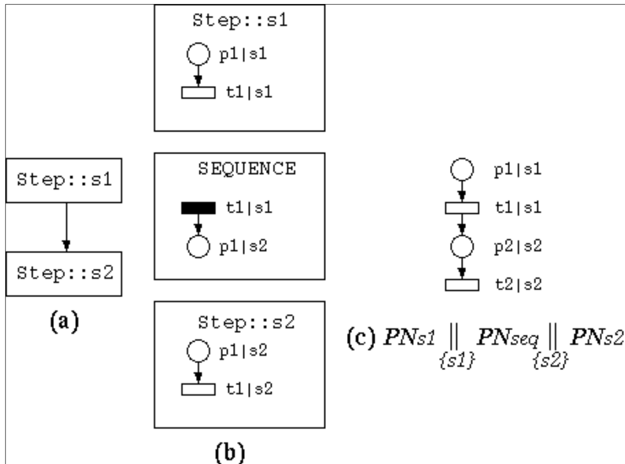

Figure 8 LGSPN patterns for Sequence, and for constructing the sequence

The pattern for a Step in Figure 7 is instantiated into a LGSPN subnet with the name of the step as a label on the starting place and on the final transition, and labels `t_host` and `r_host` (take and release "host", which is the name of the host resource for the *Step*) on the transitions before and after the timed transition for the *Step*.

The labels on places and transitions are given after the name and a vertical bar, as in "name|label", and these labels are used to compose two LGSPN subnets. Other patterns are created for the different connectors. In Figure 8 (a) a *Sequence Connector* gives a transition (labeled by the predecessor step name) and a place (labeled by the successor step name). To compose it with the two *Steps*, the transition is merged with the output transition of the predecessor *Step* (which has the same label). The labels are also merged. The place has the same label as the input place of the next *Step*, and is merged with it, as shown in Figure 8 (b) and (c). The pattern for the *Branch* uses conflicting transitions with probabilities given by the successor *Step* probabilities, as illustrated in Figure 9. The patterns for *Merge*, *Fork* and *Join* are similar.

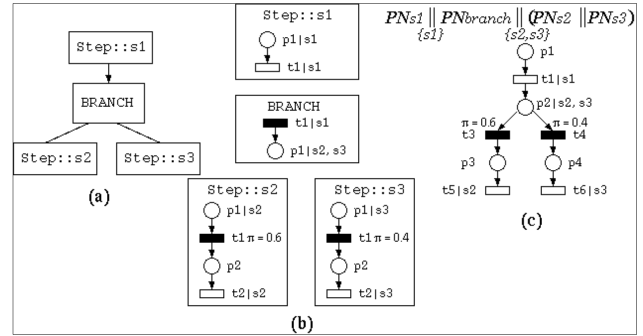

Figure 9. LGSPN patterns for Branch, and their composition

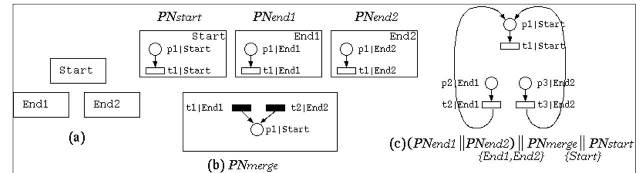

Figure 10. LGSPN patterns for Start and End

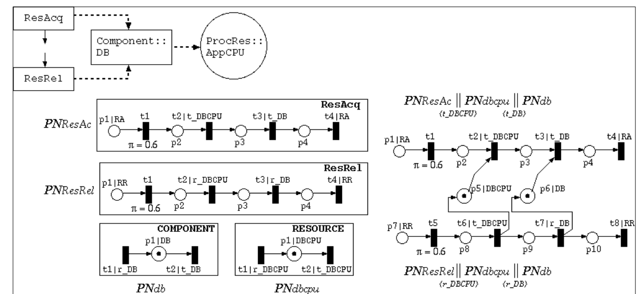

Figure 11. LGSPN pattern for composition of a logical resource.

The *Start* and the *End* patterns in Figure 10 model an applied closed workload, and are created with labels that cause all *Ends* to be joined to the *Start* of the same *Scenario*. This provides a

GSPN with a steady state cyclical behaviour. For a closed *Workload* the Start place has tokens equal to the attribute *population* of the workload of the scenario.

Resources use the pattern in Figure 11, which has:

- a place for free resources that is initially populated with a number of tokens equal to the *multiplicity* attribute of the *Resource*.
- an input transition modeling the release of the *Resource*

an output transition modeling its acquisition.

Figure 11 depicts the pattern and its composition with the host processor of a *Step*. The composition with *Resource Acquire* and *Resource Release Steps* is also shown. Composition is based on instantiating the resource pattern for every Acquire/Release pair and then merging the resource places based on their label, which is the resource name, giving a single place for each resource in the model.

The algorithm for translating a CSM *Scenario* to GSPN begins by translating all the *Resources,* then the *Start* of the *Scenario*, and the *End Steps*. The algorithm continues by translating various CSM entities:

- all the simple *Steps*, composed with their host resources
- all the *Resource Release/Acquire Steps*, composed with their *Resource* subnet

Then it composes the *Start* and *End* subnets into a result net called LGSPN, and translates each *Connector* step, composing it into LGSPN with its predecessors and successors.

The result is a GSPN for the Scenario. Figure 12 shows the Petri net that represents the CSM for the Store/Acquire Video and the CSM for the Access Control.

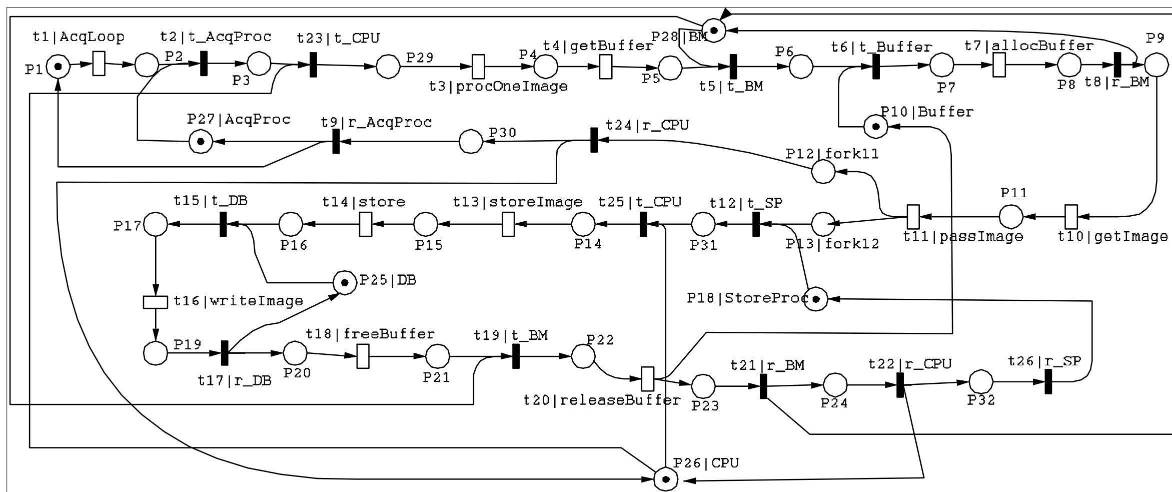

Figure 12. Petri Net produced automatically for the Video Acquisition scenario

6. FROM CSM TO QUEUEING NETWORK MODELS

The translation into ordinary (not extended) queueing networks can be carried out by applying the workload reduction technique first described by Smith [18], treating the steps in the CSM model as steps in Execution Graphs. This has not yet been automated, but the principle will be illustrated by a manual translation of the example CSM to QN.

The reduction technique in [18] gives Table 1, showing the demands for host processing (for each host) and for each External Operation. Then the external operations demands are expanded using a model for each of them, to arrive at the host demands from Table 1.

The QN model in Fig. 13 shows the movement of jobs. The service times for `AppCPU` and `DB_CPU` are given for different classes depending on the stage in the scenario. It is simpler to use the total demands given in the final row of the Table, in solving the QN as described in [18], by using standard queueing network techniques as described in [9].

Ordinary QNs do not describe simultaneous resource possession, so this model ignores logical resources such as process threads and buffers and their Resource Acquire/Release Steps. The modeling process in [18] does extend to simultaneous resources and Extended Queueing Networks, but we intend to handle such cases with Layered Queueing.

When QN creation is automated it is planned to generate the PMIL language described in [20], which should make it possible to drive the QNAP or the SPEED solvers for QNs, as described in that paper.

Table 1 Queueing Network Workload by Step, and by Server

Step	#	Applic CPU	DB CPU	LAN	Disk
		ms	ms	ops of 1.5 ms	ops of 2 ms
processImages	1				
procOneImage	$N	1.8			

getBuffer	$N	1.5			
getImage	$N	$P*1.5		$P	
passImage	$N	0.9			
storeImage	$N	1.1			
store	$N	2.0			
writeImage	$N		$B*0.9		$B
freeBuf	$N	0.2			
releaseBuf	$N	0.5			
Weighted Sum in ms.		$N*8 + $N*$P* 1.5	$N*$B*0.9	$N*$P* 1.5	$N*$B*2
Value/cycle, by server, for $N=100, $B=$P=8		2000 ms	720 ms	1200 ms	1600 ms

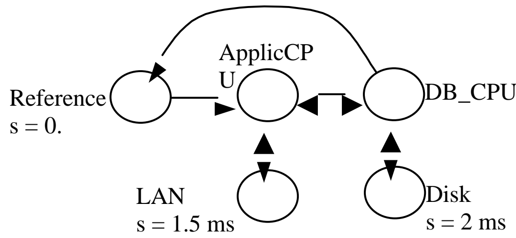

Figure 13. Queueing Network Model

7. EFFECTIVENESS

PUMA is targeted particularly to steady state performance analysis for specified scenarios. The PUMA processing architecture, with its intermediate scenario model CSM, has been shown to be effective in two ways: (a) in its power to collect and reconcile the relevant performance information from the different design sources, and (b) in the flexibility to generate different kinds of performance model using different formalisms. In PUMA, the term "scenario" includes possible variations in the path through branches and merges.

In UML, performance information for any system is recorded in at least two diagrams (behaviour and deployment); there may be multiple behaviour diagrams. The CSM creator process can search the UML (XMI) file for relevant data, exploiting the context of what it has found so far, for instance to include a Step by default even if it is not stereotyped, or to find the properties of a Component. Since it is phrased in terms very close to the SPT Profile there are few ambiguities to resolve, about the semantics of the CSM model.

Model creation has been mostly straightforward, re-using previously defined techniques for modeling from scenarios. Scenarios are a *lingua franca* for performance modeling.

This work has not addressed behaviour that is defined by state machines rather than scenarios. However, performance is almost always specified as the properties of a set of responses, which map to scenarios. Even with state machine definitions, it is necessary to project these onto scenarios; in such a case we would generate the scenarios into a CSM as a first step. There are performance-related properties which are not scenario-based in the first instance, such as the mean time to some "bad" transition, or the probability of event A occurring before event B. It is still to be explored whether these can also be mapped to a scenario-based analysis.

This paper has not addressed the systematic use of models to generate design feedback, which makes up the bottom part of Fig. 1. This will be the subject of future reports.

8. CONCLUSIONS

This paper has presented the PUMA toolset architecture and its unified model-building approach, based on a Core Scenario Model (CSM). Using an example, it has briefly described how the CSM has recently been obtained from different kinds of UML diagrams in both UML1.4 and UML2, and how it has been translated automatically to Layered Queueing models (LQN) and Petri Nets (PN), in a form that can immediately be solved by existing tools. Principles to translation to a Queueing Network model were also described, for the same example.

PUMA promises a way out of the maze of possible performance evaluation techniques. From the point of view of practical adoption, this is of the utmost importance, as the software developer is not tied to a performance model whose limitations he or she does not understand. Performance modelers are similarly freed to generate a wide variety of forms of model, and explore their relative capabilities, without having to create the (quite difficult) interface to UML. As UML is constantly changing, this can also make maintenance of model-building easier.

A limitation that this work does not overcome is the difference in the XMI produced by different UML tools. Despite the fact that XMI is an OMG standard, experience shows that, at least for now, different UML vendors introduce their own extra features in the produced XMI. The user of a particular UML tool may have to port the CSM generator to that tool. It would be interesting to create a self-configuring generator, which is sufficiently flexible to navigate these differences.

The use of non-UML specification techniques is also made easier in principle by the CSM; once a translation to CSM is made, the model-building machinery can be re-used.

This paper has explored difficulties encountered in using the "unified" model-building approach, particularly in obtaining the CSM. Mostly these confirm the wisdom of having an intermediate form, which has filtered out the mass of design information that is not needed for the performance analysis. Perhaps as a result, the experience of creating the models from CSM has revealed no serious problems.

In conclusion, PUMA is an exploration of a unified approach to modeling. So far it seems to be very successful.

ACKNOWLEDGEMENTS

This research was supported by a grant from NSERC, the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with generalized stochastic Petri nets,* John Wiley, 1995

[2] S. Balsamo and M. Marzolla. "Simulation Modeling of UML Software Architectures", *Proc. ESM'03,* Nottingham (UK), June 2003

[3] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable Petri net models," in *Proc. 3rd Int. Workshop on Software and Performance (WOSP02)*, Rome, July 2002, pp. 35-45.

[4] C. Cavenet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens, "Analysing UML 2.0 activity diagrams in the software performance engineering process," in *Proc. 4th Int. Workshop on Software and Performance (WOSP 2004)*, Redwood City, CA, Jan 2004, pp. 74-83.

[5] V. Cortellessa and R. Mirandola, "Deriving a Queueing Network based Performance Model from UML Diagrams," in *Proc. Second Int. Workshop on Software and Performance (WOSP2000),* Ottawa, Canada, September 17-20, 2000, pp. 58-70.

[6] S. Donatelli and G. Franceschinis, "PSR Methodology: integrating hardware and software models", *Proc. 17th Int. Conf. on Application and Theory of Petri Nets (Osaka, Japan*), LNCS vol. 1091, Springer, June 24-28 1996.

[7] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside, *Performance Analysis of Distributed Server Systems*, Proc. Sixth International Conference on Software Quality (6ICSQ), Ottawa, Canada, 1996, pp. 15-26.

[8] ISO/IEC 15909-1:2004 *Software and system engineering -- High-level Petri nets -- Part 1: Concepts, definitions and graphical notation*, 2004.

[9] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons Inc., 1991

[10] Ekkart Kindler *High-level Petri Nets, Transfer Syntax, Proposal for the International Standard ISO/IEC 15909-2*, Draft Version 0.3.0, April 21, 2004, at www.upb.de/cs/kindler/publications/copies/ISO-IEC-15909-2-Draft.0.3.0.pdf

[11] J. P. Lo'pez-Grao, J. Merseguer, and J. Campos, "From UML Activity Diagrams To Stochastic Petri Nets" in *Fourth Int. Workshop on Software and Performance (WOSP 2004*), Redwood City, CA, Jan. 2004, pp. 25-36.

[12] J. Merseguer, *Software performance engineering based on UML and Petri nets*, Ph.D. thesis, University of Zaragoza, Spain, March 2003.

[13] Object Management Group, *UML Profile for Schedulability, Performance, and Time Specification*, OMG Adopted Specification ptc/02-03-02, July 1, 2002.

[14] D.B. Petriu and M. Woodside, "Software Performance Models from System Scenarios in Use Case Maps", in *Proc. 12th Int. Conf. on Modelling Tools and Techniques (TOOLS 2002*), London, England, April 2002.

[15] D. B. Petriu and M. Woodside, "A Metamodel for Generating Performance Models from UML Designs", in *Proc UML 2004, v. 3273 of Lecture Notes in Computer Science (LNCS 3273*), Lisbon, Oct 2004, pp. 41-53. An extended version is to appear in the Journal of Software and Systems in 2005.

[16] D. C. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-based derivation of LQN models from UML specifications," in *Proc. 12th Int. Conf. on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation,* London, England, 2002.

[17] D. C. Petriu and C. M. Woodside, "Performance Analysis with UML," in *UML for Real*., B. Selic, L. Lavagno, and G. Martin, Eds. Kluwer, 2003, pp. 221-240.

[18] Smith, C.U. *Performance Engineering of Software Systems.* Addison-Wesley Publishing Co., New York, NY, 1990.

[19] C. U. Smith and L. G. Williams, *Performance Solutions*. Addison-Wesley, 2002.

[20] C.U. Smith, C.M. Lladó, "Performance Model Interchange Format (PMIF 2.0): XML Definition and Implementation", *Proc QEST 2004 (First Int. Conf on Quantitative Evaluation of Systems*), Enschede, Sept. 2004

[21] J. Xu, M. Woodside, and D.C. Petriu, "Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time," in *Proc. 13th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 03)*, Urbana, USA, Sept. 2003.

[22] The GreatSPN tool, http://www.di.unito.it/~greatspn