From UML to LQN by XML algebra-based model transformations

Gordon P. Gu, Dorina C. Petriu
Department of Systems and Computer Engineering
Carleton University, Ottawa, ON, Canada
{pgu|petriu}@sce.carleton.ca

ABSTRACT

The change of focus from code to models promoted by OMG's Model Driven Development raises the need for verification of non-functional characteristics of UML models, such as performance, reliability, scalability, security, etc. Many modeling formalisms, techniques and tools have been developed over the years for the analysis of different non-functional characteristics. The challenge is not to reinvent new analysis methods for UML models, but to bridge the gap between UML-based software development tools and different kinds of existing analysis tools. Traditionally, the analysis models were built "by hand". However, a new trend is starting to emerge, that involves the automatic transformation of UML models (annotated with extra information) into various kinds of analysis models. This paper proposes a transformation method of an annotated UML model into a performance model. The mapping between the input model and the output model is defined at a higher level of abstraction based on graph transformation concepts, whereas the implementation of the transformation rules and algorithm uses lower-level XML trees manipulations techniques, such as XML algebra. The target performance model used as an example in this paper is the Layered Queueing Network (LQN); however, the transformation approach can be easily tailored to other performance modelling formalisms.

General terms

Design, Performance.

Keywords

Software Performance Engineering, UML, performance profile, automatic model building, model transformations, XML, XMI, LQN.

1. INTRODUCTION

Model Driven Development (MDD), the new approach to software development proposed by OMG, promotes the idea that software development should be based on models throughout the entire software lifecycle. UML and other OMG standards play an important role in MDD. This change of focus from code to models raises the need for verifying non-functional (also known as extra-functional) characteristics of UML models, such as performance, reliability, scalability, security, etc. Over the years, many modeling formalisms, techniques and tools have been developed for the analysis of different non-functional characteristics. The challenge is not to reinvent new analysis methods for UML models, but to bridge the gap between UML-based software development tools and different existing analysis tools. Traditionally, the analysis models were built "by hand" by specialists in the field, then solved and evaluated separately with known tools. However, with the change of focus on models brought by MDD, a new trend is emerging for the evaluation of non-functional properties of software systems under develop-ment, which involves the automatic transformation of UML models into different analysis models. For instance, such an approach was used for performance analysis of UML models in [6][15][21], and for security characteristics verification in [7].

Different kinds of analysis techniques may require additional annotations to the UML model to express, for instance non-functional requirements and characteristics. OMG's solution to this problem is to define standard UML profiles for different purposes. Two examples of such profiles are the "UML Profile for Schedulability, Performance, and Time" (SPT) [11] and "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" [12].

Translations from UML into different performance models have been surveyed in [1], such as translations to queueing models in [18][5], to LQN in [6][15], to stochastic Petri nets in [3][10], to stochastic process algebra in [4], directly to simulation in [2]. More recently, a transformation framework from multiple input design models into different performance models was proposed in [21]. However, so far the transformation process itself has been ad-hoc, tailored to the source and target formalisms.

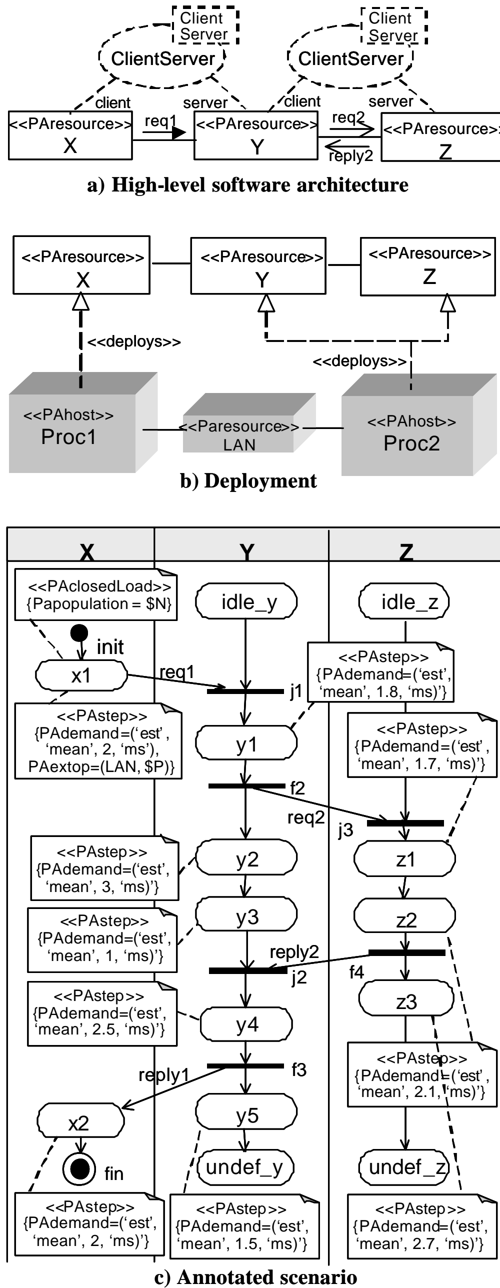
The contribution of this paper is to formalize the transformation process from annotated UML design models to performance models, in order to make it more modular, easier to apply and specialize it for different performance models. The mapping between the input (source) model and the output (target) model is defined at a higher level of abstraction (i.e., at the metamodel level) by using graph transformation concepts, whereas the implementation of the transformation rules and algorithm is done at the XML level, using lower-level XML trees manipulations techniques, such as XMLgebra and XACT proposed in [8][9]. The target performance model used as an example in this paper is the Layered Queueing Network (LQN); however, the transformation approach can be easily tailored to other performance modelling formalisms.

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 11-14, 2005, Palma de Mallorca, Spain.
Copyright 2005 ACM 1-59593-087-6/05/0007 …$5.00.

2. BACKGROUND

2.1 Annotated UML design models

The input to the proposed transformation method is a UML 1.4 design model annotated with performance information according to the SPT profile, as illustrated in Figure 1 by a very simple example of a 3-tier client/server model. The structure of the performance model is obtained from: a) high-level software architecture showing the concurrent components and their relationships, possibly through design patterns as in Figure1.a, and b) deployment information describing the allocation of software components to hardware devices, as in Figure1.b.



a) High-level software architecture

b) Deployment

c) Annotated scenario

Fig.1. Annotated UML model of a 3-tier client/server

The behaviour of the performance model is derived from key scenarios modeled as UML activity (as in Figure 1.c) or interaction diagrams, showing the activities executed by different components and the flow of control/data in the system.

The activity diagram from Figure 1.c represents a key scenario of the 3-tier client-server model. We make the assumption that each software process from the high-level software architecture shown in Figure 1.a is represented by a separate "swimlane" in the activity diagram, which contains all the steps performed by that process. We also assume that the transitions crossing the swimlane boundaries represent an exchange of messages (signals) between concurrent components (named cross-transitions in the paper) even though UML 1.4 does not imply this interpretation. If we need to model explicitly the content of a message, we can use an ObjectFlowState (not shown in the example from Figure 1.c for the sake of simplicity). In order to simplify the transformation, we also assume that the cross-transition name is identical to the message name shown in the high-level architecture (Figure 1.a). The type of messages, either synchronous or asynchronous, is also denoted in the collaboration diagram. We made the assumption that the instance that initiates the scenario starts at the initial state of the activity diagram and ends at its final state. All the other components are assumed to have a cyclic behaviour, waiting in a state named "idle" to receive their first signal that triggers them into action. At the end of the scenario, these components will return to an undefined state by default (which may or may not be the idle state). By collecting the partial behaviours of a component from different scenarios, one can build the complete behaviour for every component; however, this is beyond the scope of the paper. In the transformation to performance model presented here, the idle and undefined states serve as begin/end indicators inside a partition, but do not represent actual scenario steps and will not be translated into the IM model.

Since we assume that processes (component) represented in swimlanes are concurrent, we use the following convention inspired from Petri Nets to represent the sending/receiving of messages (signals) between components. On the sender's side, the activity sending a message (be it synchronous or asynchronous) is followed by an explicit forking: one thread for the continuing execution of the sender, and the other thread for the message just sent (e.g., fork `f2` from Figure 1.c). On the receiver's side, the message is accepted through the joining of the receiver's thread with the message thread (e.g., join `j3`). A synchronous communication is composed from a request and a reply, which can be represented by two related messages (e.g., `f2` and `j3` represent a request from `Y` to `Z`, while `f4` and `j2` represent the corresponding reply). Note that, in Figure 1.c, after sending a request to `Z`, the sender `Y` continues its execution with the activities `y2` and `y3`, and which will accept the reply. In some cases, the sender of a synchronous request (e.g., `X` in Fig. 1.c) may block immediately after sending the request, and will continue only when the reply arrives. In this case, a simplified representation may be used, in which the "sending" fork for the request and the "receiving" join for the reply are omitted.

The hardware resources represented as nodes in deployment diagrams are stereotyped either as `<<PAhost>>` for processors, or `<<PAresource>>` for other devices (disk, network, etc). Concurrent processes participating in scenarios are also stereotyped as `<<PAresource>>`. A scenario is composed of steps

stereotyped as `<<PAstep>>`, which are either the effect of messages in UML interaction diagrams, or activities in UML activity diagrams. Among the attributes of a `<<PAstep>>` is `PAdemand` that gives the "host demand" as a `PAperfValue` type. For example, the step `y2` has an assumed mean CPU demand of 3 ms expressed as:

```
PAdemand=('asmd', 'mean', (3, 'ms'))
```

The first step of a scenario has the scenario workload information associated to it: in this case a closed workload with `$N` users. More details on how to use the SPT performance annotations can be found in [16].

The proposed transformation reads the input UML model in XML format, produced by an existing UML/XMI tool according to the standard XML Metadata Interchange (XMI).

2.2 XMLgebra

XMLgebra was introduced in [9] to resolve the problem of static validation of dynamically constructed XML documents used in web services. It is a theoretical foundation on top of which XML becomes a first-class data type in any modern programming language (e.g., Java). It was implemented in the XACT framework [8], which allows programmers to manipulate XML templates as first-class data types in a Java program. The XMLgebra operations are based on the XPath and DTD standards. In this work, XMLgebra is used to express higher-level transformation rules from an input to an output model.

In XMLgebra, the processing of documents is done in terms of XML templates [8][9]. An XML template (as shown in Figure 3) is a well-formed XML fragment, containing named gaps that may appear in place of elements and attributes.

Definition. An XML template is defined as $t = (dom_t; lab_t; (\lambda^a_t)\ a \in A)$ where:

- Σ = a finite set of vertex labels,
- A = a set of attribute names,
- G = a set of gap names,
- N^* = a set of strings of natural numbers,
- D = an infinite (recursively enumerable) domain of values,
- dom_t = a finite template domain over N^*,
- $lab_t: dom_t \rightarrow \Sigma$ is a labeling function,
- for every attribute name $a \in A$, $\lambda^a_t: dom_t \rightarrow D \cup G$ is a partial attribute value function.

The set of all such templates is denoted as $T_{(\Sigma;A;G)}$.

The domain dom_t encapsulates the structure of an XML template (tree); every tree vertex ν is represented by a value from N^* converted to a string that encodes information similar to XPath (see Figure2).

Let $\nu \in dom_t$ be a vertex corresponding to a given XML node; the i'th child of ν has the string value $w = \nu{\times}i$. When assigning ν values,

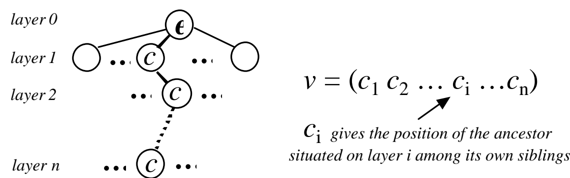


Figure2. Notation for a vertex $\nu \in dom_t$

the order in which different elements appear in the XML document is respected. The labeling function is defined to attach a label from Σ to a vertex in dom_t. Attribute value functions λ^a_t are defined to extract the value of an attribute a from the XML element v given as a parameter.

The character data of an XML template is listed in sequence, each element representing a node labeled with $PCDATA \in \Sigma$. The character sequence is represented by a special attribute named $PC \in A$, together with a partial function λ^{PC}_t mapping PC to the corresponding data value at each $PCDATA$ node. The *gaps* of an XML template are represented using nodes labeled with another special symbol $GAP \in \Sigma$. A gap has a name represented by a special attribute named $GN \in A$, together with a partial function λ^{GN}_t mapping GN to the corresponding gap name for each GAP node.

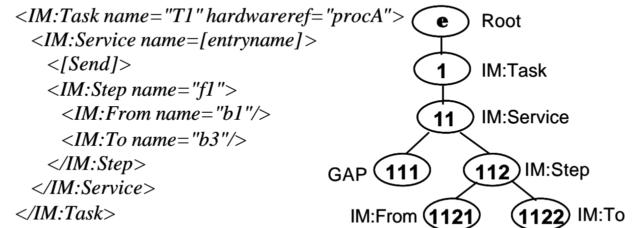


Figure 3. Example of a XMLgebra template

Figure 3 shows an example of template fragment where `[Send]` is a *template GAP* and `[entryname]` an *attribute GAP*. The label function lab_t maps dom_t to Σ and the attribute functions returns the value of an attribute.

The detailes of $T_{(\Sigma;A;G)}$ are as follows:

$dom_t = \{\mathbf{e};1;11;111;112;1121;1122\}$,

Σ={ Root; IM:Task; IM:Service; GAP; IM:Step; IM:From; IM:To }

A={name; hardwareref; GN}

G = {entryname; Send}

lab_t = [**e** → ROOT; 1 → IM:Task; 11 → IM:Service;111 → GAP; 112 → IM:Step; 1121 → IM:From; 1122 → IM:To]

$\lambda_t^{name}(1) = T1$; $\lambda_t^{hardwareref}(1) = procA$;
$\lambda_t^{name}(11) = entryname$; $\lambda_t^{GN}(111) = Send$; $\lambda_t^{name}(112) = f1$;
$\lambda_t^{name}(1121) = b1$; $\lambda_t^{name}(1122) = b3$.

Using strings to represents the values in dom_t has the drawback that it fails to handle unambiguously very large templates. For instance, a string expression "1111" may be interpreted as {"1","1","1","1"}, {"1","1","11"}, {"1","11", "1"}, etc., which are different vertices. To avoid such confusion, we proposed to use an array of integers for dom_t values, instead of strings. This small change affects the formal definition of all the operations for template manipulations, so we use the name eXMLgebra to indicate this extension.

As already mentioned, eXMLgebra contains a number of operation (functions) for template manipulation. Only the effects of some such operations are briefly described here. Figure 4.a illustrates the effect of operation *select*, which uses the input template $t \in T_{(\Sigma,A,G)}$ and the input XPath v to extract and return the subtree rooted at v

as a new template t', while the original template t remains unchanged. XML templates can be constructed using the special *tplug* operation, illustrated in Figure 4.b, which inserts an existing XML template into the gaps of another XML template. A similar operation named *splug* (not shown here) inserts string values into gaps, be they nodes or attributes. Figure 4.c shows the effect of operation *remove*, which removes a node from a template, but not its corresponding subtree. In fact, the children of the removed nodes are brought one level up. It is also possible to insert gaps (nodes or attributes) in desired places by using *gapify* operations (not shown here).

3. TRANSFORMATION CONCEPTS

3.1 General Approach

The proposed transformation method accepts an XML file

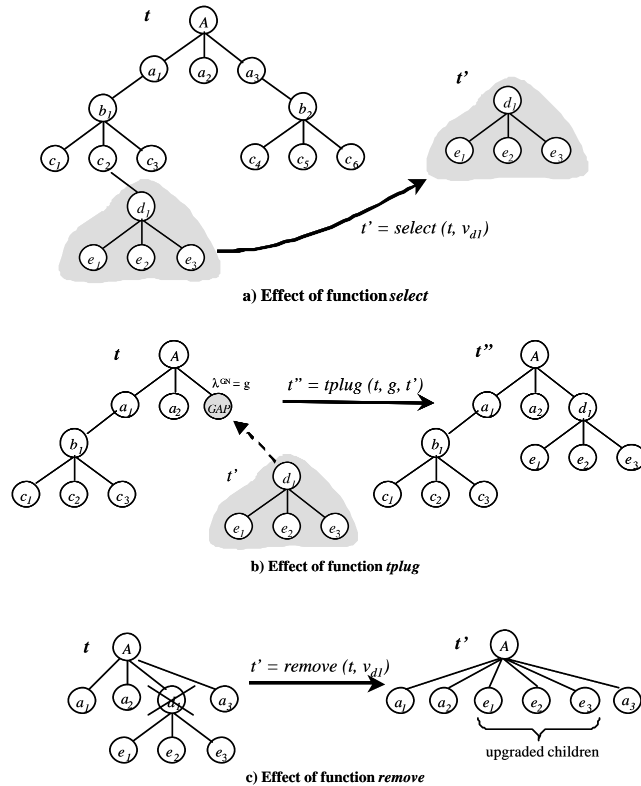


Figure 4. Some template manipulation operations

representing the input model that is compliant with the input DTD/schema, and generates an output XML tree that is compliant with the output DTD/schema. The input and output DTD (schemas) describe formalisms from different domains, so the semantic gap between them may be quite important. Optionally, the output XML data structure may be further converted into a text file, if the analysis tool requires a text format. However, this latest step is simpler and will not be discussed in the paper. The challenge is to define the mapping and to bridge the semantic gap between the input and output models at a higher level of abstraction. This step is based on graph transformation concepts[19][20], which have proven to be powerful enough for such applications in previous work by the authors of this paper [14][15][6].

The essential idea of *graph grammars* or *graph rewriting systems* is that they are generalization of the *string grammars* that are used in compilers. The terms "graph grammars" and "graph rewriting systems" are often considered synonymous. However, the first is a set of production rules that generates a language of terminal graphs and produces nonterminal graphs as intermediate results, whereas the second is a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs, without distinguishing between terminals and nonterminals graphs. The main component of a graph grammar is a finite set of production rules. A production is a triple (L,R,E), where L and R are graphs (the left-hand side and right-hand side, respectively) and E is an embedding mechanism. Such a production rule can be applied to a host graph H as follows: when an occurrence of L is found in H, it is removed end replaced with a copy of R; finally, the embedding mechanism E is applied to attach R to the remainder of the host graph H

What is new in this paper is that, although the mapping between the input and output model are defined at the metamodel level, the detailed definition and implementation of the transformation rules and algorithms happens at the XML level, using XML tree manipulations based on eXMLgebra. In this way, we can take advantage of numerous XML technologies and tools that have been developed in the past year. More specifically, our implementation of the proposed method uses XSLT.

The role of each transformation rule is to map a concept from the input model, represented as a template compliant to the input DTD, to an output concept represented by an output template. Also, a rule defines how to compute the output node attributes based on the input nodes. Besides the rules, a transformation algorithm is needed to decide in what order to invoke the transformation rules over a given input tree for generating output subtrees, and how to "glue" these subtrees together to construct the complete output tree. Conceptually, the "gluing" of subtrees is a label-based process, where the labels are node attributes of the output subtrees. More details are given in Sections 4 and 5.

3.2 Two-step transformation from UML to LQN

One of our goals is to develop a generic modular transformation method, easy to specialize for accepting several kinds of input models (such as UML 1.4 and UML 2.0) and for generating different kinds of output models. (We have tried so far LQN and CSIM-based simulation, but the later is not discussed in this paper). The PUMA project [21] makes a strong argument for solving this type of *N-by-M* problem by introducing a common intermediate format named the Core Scenario Model (CSM) [13]. PUMA's CSM captures the essence of performance specifications from a UML design as expressed in the SPT Profile [11] and strips away the design detail that is irrelevant to performance analysis. In this work we have adopted a similar approach.

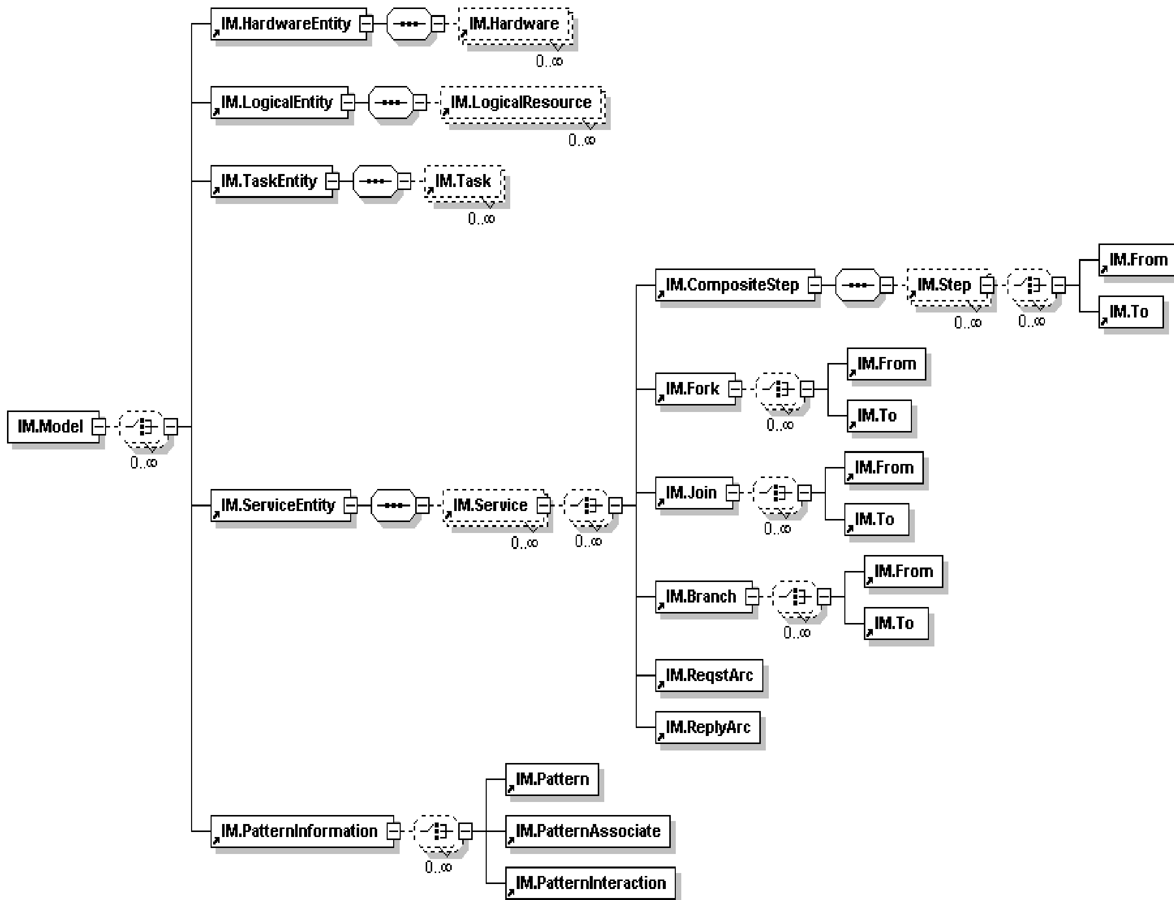


Figure 5. XML Schema of the Intermediate Model

Our proposed Intermediate Model (IM) mirrors the Performance Subprofile concepts, was developed in parallel with CSM and has many of the same features.

The Performance Subprofile from SPT [11] describes a domain model that contains the basic abstractions used in performance analysis, including scenarios, resources and workload. Scenarios define response paths through the system, and can have QoS requirements such as response times or throughputs. Scenarios are executed by either closed or open workloads. Each scenario is composed of scenario steps that can be joined in sequence, loops, branches, forks and joins. A scenario step may be an elementary operation at the lowest level of granularity, or may be a complex sub-scenario composed of many basic steps. Each step has a mean number of repetitions, a host execution demand, other demands to resources and its own QoS characteristics. Resources are another basic abstraction, and can be active or passive, each with their own attributes.

Figure 5 shows the IM schema. Its top node, *IM:Model* contains *IM:HardwareEntity* (hardware devices and processors), *IM:Task* (software components), *IM:Service* (scenario), and smaller elements such as *IM:Step*, *IM:Join*, *IM:Fork*, *IM:Branch*, *IM:merge*, *IM:ReqestArc*, *IM:ReplyArc*, etc. IM can also model logical resource, patterns and other supporting information.

The main difference between CSM and IM is that the first is a scenario-based model (i.e., the steps are grouped in accordance with the scenario they belong to, regardless of who executes them) whereas IM is task-based (i.e., the steps are grouped by tasks). IM was developed separately from CSM only for practical reasons, to allow for independent work by different researchers in the same larger group. We have not migrated IM to CSM yet, because the SPT Profile, the very basis of both of them, will undergo a substantial upgrade in the near future. We foresee that IM will be eventually replaced with CSM.

Similar to the PUMA approach, the transformation from UML to LQN discussed in this paper is done in two steps:

1. Extract the relevant UML model information and. performance annotations from the XMI input file obtained from an UML tool, and generate the corresponding Intermediate Model. IM is another XML file that contains only the information required to build a performance model, filtering out a lot of UML model details unrelated to performance.

2. Generate a LQN model in XML format from the IM obtained in the previous step. After the XML tree is generated, it is very easy to traverse it and produced the textual LQN format expected by some of the LQN tools.

A concrete example of a partial IM model is given in Figure 6. It corresponds to the scenario steps from Figure 1.c executed in the swimlane `Z` by the concurrent process with the same name. In IM the steps appear in the same sequential order as in the activity diagram swimlane (i.e., the sequence relationship between steps is implied by their position). Only fork/join and branch/merge are represented explicitly as IM nodes.

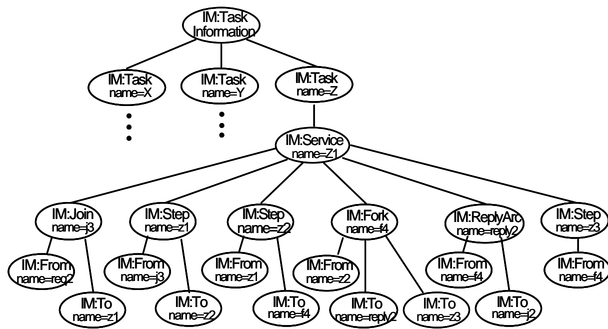


Figure 6. IM submodel example

4. FROM UML TO IM

The mapping of the main concepts from UML to IM and LQN is shown in Table 1. The mapping is not always straightforward, and depends on certain model-wide conditions. For instance, all UML activity diagram transitions are modelled by the same metaclass, but only some of them will have a special meaning in our transformation (i.e., the cross-transitions discussed in section 2.1).

Table 1. Mapping of modelling concepts

UML Model	IM	LQN
Node <<PAresource>>	Hardware Resource	Hardware Device
Node <<PAhost>>	Processor	Processor
Class/Object/ Component <<PAresource>>	Logical Resource	Task or logical resource
Partition	Logical Resource	Task
Partition containing Initial/End Pseudostate	Task containing Initial/End Step	Reference Task
Cross-transition	Request and/or Reply Arc	LQN request arc
ActionState or SubactivityState <<PAstep>>	Step	
Selected group of States	Service	Entry
Selected group of States	Selected set of Steps	Activity or Phase
Join/Fork Pseudostate	Join/ Fork	"AND" Join/Fork
Branch/Merge Pseudostate	Branch/ Merge	"OR" Join/ Fork

Another example of complex mapping requires the identification of groups of steps that will be mapped to LQN entries, phases and activities, as discussed in section 4.1. All these mappings are expressed as transformation rules, and each rule applies in certain conditions.

4.1 Transformation Rules

Some transformations rules from UML to IM are simple to understand, as they map one UML concept to a corresponding IM concept (e.g., *UML:Node* to *IM:HardwareResource*, *UML:Object* to *IM:LogicalResource*; *UML:ActionState* to *IM:Step*, etc.) Even so, one-to-one mappings at the conceptual level do not necessarily translate into a simple node-to-node transformation at the XML tree level, as each modelling concept may be represented by a XML template, not just by a node.

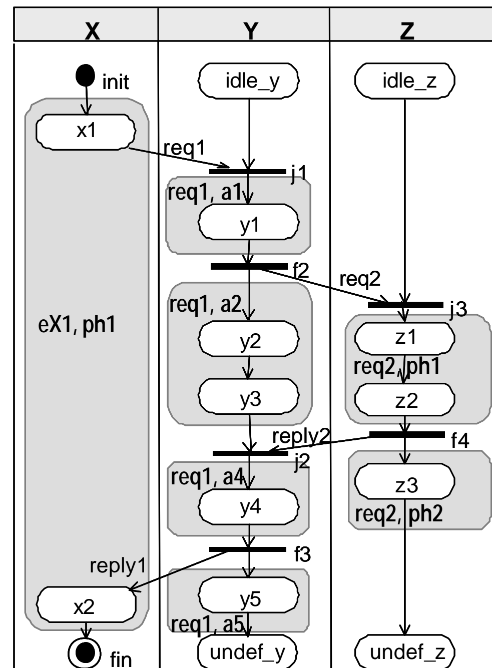


Figure 7. Aggregation of scenario steps

A transformation rule becomes more complex if it represents a one-to-many or many-to-one mapping. To understand the need for such mappings, let us consider the activity diagram from Figure 1.c redrawn in Figure 7, with shaded areas representing groups of activities (i.e., scenario steps) that will be eventually aggregated together in the LQN domain. For instance, the group of steps executed by a process in response to a service request will become an *LQN:Entry*, which in turn may be split into *Phases* or *Activities*. In an entry, the subgroup of steps executed in a single thread of control between the receiving of a service request until the sending of the reply will generate a single LQN element, namely *Phase1* of the *LQN:Entry* modelling the service. All the steps executed by the same objects after sending the reply until it reaches the last state will generate *Phase2* of the same *LQN:Entry*. For example, steps `z1` and `z2` will be grouped together to generate `phase1` of entry `req2`, whereas step `Z3` will generate `phase2` (see also Figures 9 and 10). If, however, a forking occurs between the receiving of a

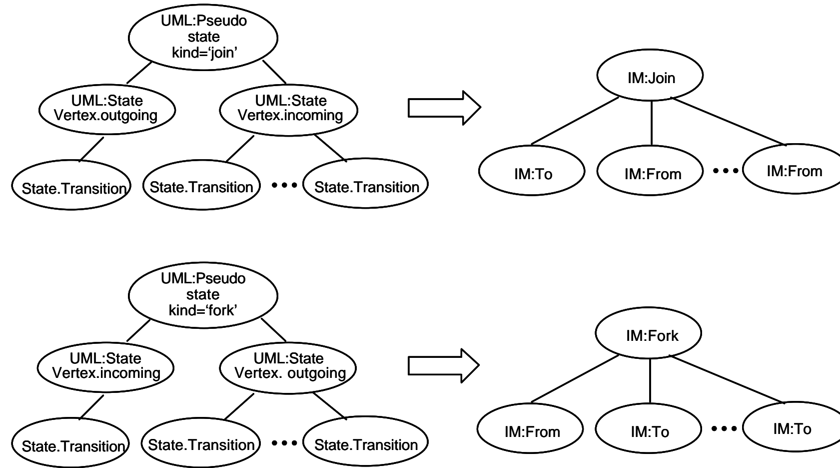
request and the sending of the reply, the LQN feature *LQN:Activity* will be used instead of *LQN:Phase*. For example, the shaded areas from Figure 7 show the groups of steps contained in swimlane `Y` that generate activities `a1` to `a5`. More explanations on how these groups are generated can be found in [15]. However, the point we are trying to make here is that the mapping from one domain to another is not always straightforward. The mapping depends usually on conditions that may be simple or complex, local or global. The conditions for applying a given transformation rule need to be expressed either as a part of the transformation rule itself, or to be included in the transformation algorithm that controls the applications of the rules. In this work, both the rules and the transformation algorithm are expressed at the XML level, with the help of eXMLgebra.

Figure 8 shows an example of a transformation rule that is mapping synchronizations bars from the activity diagrams (i.e., *UML:Pseudostate*) into *IM:Fork* or *IM:Join* connectors. The left-hand side XML template contains a *Pseudostate* of kind *"fork"* or *"join",* which has a number of *UML:State.Vertex.outgoing* and *UML:State.Vertex.incoming*. Each *State.Vertex* has one or more children of type *State.Transition*, each one with an *xmi.idref* attribute that refers to an activity diagram transition.

Two versions of the left-hand side template are shown at the top of Figure 8, one for *"join"* and the other for *"fork"*. The transformation rule definition given below indicates the conditions for choosing one case or the other. The symbol "→" expresses a direct node-to-node mapping. For instance:

UML:Pseudostate(xmi.id, name, visibility, isSpecification, kind)
→ *IM:Join (name, taskname)*

indicates that a node with four attributes labeled *Pseudostate* from the UML domain is mapped to a node with two attributes labeled *Join* from the IM domain. The transformation rule defines also the attribute conversions by λ^a functions defined in section 2.2.



UML:Pseudostate(xmi.id, name, visibility, isSpecification, kind)
→ *IM:Join(name, taskname)* **if** $\lambda^{kind}(v_{pseudostate}) =$ 'join' ;
→ *IM:Fork(name, taskname)* **if** $\lambda^{kind}(v_{pseudostate}) =$ 'fork' ;
UML:StateVertex.outgoing → *null;*
UML:StateVertex.incoming → *null;*
BehavioralElements.StateMachines.Transition (xmi.idref)
→ *IM:To(name)* **if** $(v_{state_machine.transition}) \in children(v_{statevertex.outgoing})$;
→ *IM:From(name)* **if** $(v_{state_machine.transition}) \in children(v_{statevertex.incoming})$;
$\lambda^{name}(v_{fork}) = \lambda^{name}(v_{pseudostate})$;
$\lambda^{name}(v_{join}) = \lambda^{name}(v_{pseudostate})$;
$\lambda^{taskname}(v_{fork}), = \lambda^{name}(v_{partition}), \exists \lambda^{name}(v) = \lambda^{name}(v_{from}) \wedge v \in descendants(v_{partition}) \wedge v_{from} \in children(v_{fork})$;
$\lambda^{taskname}(v_{join}), = \lambda^{name}(v_{partition}), \exists \lambda^{name}(v) = \lambda^{name}(v_{to}) \wedge v \in descendants(v_{partition}) \wedge v_{to} \in children(v_{fork})$;
$\lambda^{name}(v_{from}) = \lambda^{name}(v \mid v \in descendants(v_{compositstate.subvertex}) \wedge \lambda^{xmi.idref}(v) =$
$\lambda^{xmi.id}(v_{transition} \mid w_{transition.source} \in descendants(v_{transition}) \wedge \lambda^{xmi.idref}(w) = \lambda^{xmi.idref}(v_{state.transition})))$;
$\lambda^{name}(v_{to}) = \lambda^{name}(v \mid v \in descendants(v_{compositstate.subvertex}) \wedge \lambda^{xmi.idref}(v) =$
$\lambda^{xmi.id}(v_{transition} \mid w_{transition.target} \in descendants(v_{transition}) \wedge \lambda^{xmi.idref}(w) = \lambda^{xmi.idref}(v_{state.transition})))$;

Figure 8. Transformation rule example

4.2 Execution Sequence in IM

The execution sequence shown in the UML activity diagram is not explicitly expressed by the individual transformation rules. However, IM needs to generate in order from left to right the steps that are executed sequentially, as there is no sequence connector between IM steps, only branch/merge and fork/join connectors. More specifically, an *IM:Service* template must reflect the execution step sequence from the UML activity diagram (see Figure 6 as an example). The generation of a *IM:Service* template is based on the information contained in the corresponding *UML:Partition* template.

The algorithm shown below determines what kind of gaps to generate in the output template according to the scenario steps from the input *UML:Partition* template. The algorithm traverses each element in the *UML:Partition* template starting from the initial vertex of each partition. It creates a corresponding gap in the *IM:Service* template and plugs in the *IM:Step* template that was generated previously by the appropriate transformation rule. The following UML scenario steps generate *IM:Steps*: a) activities sterotyped as `<<PAstep>>` with a defined service time; b) fork and join *Pseudostates*, c) branch and merge *Pseudostates,* and d) cross-transitions representing a message exchange between concurrent processes. In the IM domain, each of the plugged templates contains also information on its predecessor and successor nodes, so that the generated IM maintains the scenario sequence described in the UML activity diagram.

```
construct(T_task, T_collection, T_patition){
  v = first(T_patition);
  t_1 = T_service ; t_2 = T_patition ; t_3 = T_collection ;
  T_task = traverse(v, t_1, t_2, t_3)
}
traverse(v, t_1, t_2, t_3 ) {          // recursive function
  if ( descendants_t2(v) = {0})
         return;   // this vertex has
                   // no outgoing transition
 if(ν ∉ t_2) return; //this vertex is not in this partition
 if (lab_t2(ν) = s| s ∈ S ) {
   g = lab_t2(ν);
   gapifyAChild(t_1, u|lab_t1(u) = 'Service', g);
   t = select(w| w ∈ t_3, λ_t3^name(w) = λ_t2^name(v));
   tplug(t_1, g, t);
 }
 ∀ v_i| v_i= descendants_t2(v) ^ lab_t2(v_i) = 'To') {
                  // outgoing transition
   if(v|λ_t3^name(v) = λ_t2^target(v_i) ^ v ∉ t_2)
                  // this transition crosses boundary
       g = lab_t2(v_i);
       gapifyAChild(t_1, v|lab_t1(v) = 'Service', g);
       t = select(w| w ∈ t_3, λ_t3^name(w) = λ_t2^name(v));
       tplug(t_1, g, t);
       }
   else{        //a local transition
     v_t =( v|λ_t3^name(v) = λ_t2^target(v_i) ^ v ∈ t_2);
     traverse(v_t, t_1, t_2 );
     }
 }
 if(∃ v| v∈ t_2 ^ v∉ t_1) traverse(v, t_1, t_2, t_3 );
}
```

4.3 Algorithm for gluing the output IM tree

From the application of transformation rules we obtain a set of templates $\{T\} = \{T_1, T_2, T_3, \ldots, T_i\}$ where T_i is a collection set that contains the required fragments for constructing the intermediate file template. Also, the transformation needs the DTD for IM, T_{dtd}, to check whether all nodes v are defined in T_{dtd}. The construction of the output IM tree becomes a function of $\{T\} \times T_{dtd} \rightarrow T_{im}$. In other words, we use the template fragments in $\{T\}$ to construct the intermediate model template in accordance with a given DTD template. The "gluing" algorithm is given below:

```
T_im= {ε};
∀ v | v ∈ T_dtd
   f(v);

f(v){                    // recursive function
{S_t} = {w| w∈ {T}, lab_w(1) = lab_dtd(v)};
if {S_t} = {0} return;
else ∀ S_i ∈ {S_t}
  p_i = findparent({T},S_i);
  if p_i ∈ T_im {
    if S_i ∈ T_im
      return;
    else {
       g = λ^name(S_i);
       gapifyAChild(T_im, p_i, g);
      tplug(T_im, g, S_i);
     }
   }
  else      //p_i ∉ T_im;
    f(p|lab_dtd(p) = lab_t{p_i},p ∈ T_dtd ^ p_i ∈ {T});
}
```

This algorithm provides a generic method to construct an output template from a given set of template fragments and an input DTD template. All the smaller templates in $\{T\}$ must be compliant with the given input DTD. The fragments in $\{T\}$ can either be a single vertex or a subtree. The function $f(v)$ is a recursive function. It takes a vertex from the T_{dtd} and tries to match it with template fragments in $\{T\}$. If matches are found, it takes the parent vertex of the matching template to check if the parent is already in the new template T_{im}. If the parent vertex exists and the matching child does not exist, a gap is created and the matching vertex is plugged into the new template T_{im}; otherwise the algorithm returns. If the parent vertex is not found in the new template, then use the parent vertex as the matching vertex and repeat the steps. Moreover, the *findparent({T}, v)* function returns a vertex whose attribute name is the same as the attribute "pname" of v, e.g., $\lambda^{name}(w) = \lambda^{pname}(v)$.

Note that the IM subtree from Figure 6 is obtained by the transformation described in this section.

5. FROM IM TO LQN

This section describes how to transform IM to LQN. Before getting into the transformation details and rule definitions, it is necessary to understand the mappings between the IM and LQN models, as shown in Table 1. It should be noted that, while the mapping from UML to IM is independent of the target performance model, the mapping from IM to a certain performance model is highly specific to that performance model. Our goal is to limit the dependency of

the proposed algorithms on the specific model details as much as possible, for achieving reusability and flexibility.

5.1 IM → LQN Rules

The first step is to define the mapping rules between the two domains. Each rule has a left-hand side template from the IM domain, and a right-hand side template from the LQN domain. The following list gives only the root node labels for the input and output templates.

```
IM:TaskInformation  → LQN:TASKINFO;
IM:HardwareEntity → LQN:P;
IM:Hardware → LQN:P_DECL;
IM:LogicalEntity → LQN:T;
IM:LogicalResource → LQN:T_DECL;
IM:Task → LQN:TASK;
IM:Service → (LQN: PH_DECL,
            LQN: A_DECL(LQN: ACTIVITY(LQN:ACONNECTION)));
IM:Step → (LQN:PHASE, LQN:AS);
IM:Fork → (NULL, LQN:SPLIT);
IM:Join  → (NULL, LQN:JOIN);
IM:ReqstArc → (LQN:PCALL, LQN: ACALL);
IM:ReplyArc → (NULL, NULL);
IM:Branch → (NULL, LQN:SPLIT);
```

As mentioned in section 4.2 and illustrated in Figure 7, one of the most challenging rules from IM to LQN has to deal with identifying groups of *IM:Steps* and mapping them to *LQN:Phases* or *LQN:Activities*, depending on rather complex conditions. This problem was inherited from the UML model, but was not solved in the UML-to-LQN transformation, as each *IM:Step* corresponds exactly to one UML step. Therefore, the problem of grouping the *IM:Steps* and mapping them to the LQN domain (entries, phases and activities) has to be solved now.

The solution is encapsulated in the following rule:

```
IM:Service → (LQN:PH_DECL, LQN:A_DECL (LQN:ACTIVITY
                                  (LQN:ACONNECTION)));
```

The choice is further delegated to the transformation function *select-apply*, which takes a vertex from the input template (in the IM domain), verifies all the application conditions and returns the corresponding vertex, which is the root of the output template (in the LQN domain). The purpose of the *select-apply* function is even more far-reaching. Its goal is to encapsulate the verification of all conditions for rule applications, which are specific to the performance target model, leaving the transformation algorithm that invokes *select-apply* and controls the application of the IM→LQN transformation rules as generic as possible. The pseudo-code for the *select-apply* function used in the transformation from IM to LQN is given below:

```
select-apply(v)
{
  if (v Ï V) return null;
  if (lab(v) = 'Service'){
   if ($ v_i,/ v_i Î descendants(v), u=lab(v_i)=Fork
          ^ $ u_i,/ u_i Î descendants(u), lab(u_i)? ReplyArc)
      return w=lookup(v(1))     ;// it is an activity
   else return w=lookup(v(0))  ;// it is a phase
  } // end if
  else{
    if (p = parentService(v)==null)
      return w=lookup(v);
   else{
      if (lab(select-apply(p) = 'A_DECL')
        return w=lookup(v(1));
      if (lab(select-apply(p) = 'P_DECL'){
        if(lab(v) = 'Step' ^ $ v_c,/ v_c =children(v)
           ^ lab(v_c) ='From' ^l^name(v_c) = l^pname(v)) {
           while(u = p.i Î Tim){
             if(lab(u) = 'Step' ^ l^pname(u)= l^pname(v))
               st = st + l^servicetime(u);
            i=i+1;
            } // end while
           l^servicetime(w)=st;
            if(l^name(v_c) = 'Join') l^name(w)=phase1;
            if(l^name(v_c) = 'Fork') l^name(w)=phase2;
            return w;
        }
        else
           return null;
       return w=lookup(v(0));
      }
    }
  }
} // end select-apply
```

The *lookup* function returns an element from the right side of the rule table. The *select-apply* function takes an input node and returns one of the following: null, node or tree, depending whether a match with the left-hand side of a rule from the set IM→LQN rules was found or not. The choice of whether to convert an *IM:Service* template to *LQN:Phases* or *LQN:Activities* is made as follows.

The IM input model is traversed to determine whether a fork operation, which is not a reply, exists. A conversion to "phase" is applied if there is no such a fork, otherwise a conversion to "activity" is considered. The fork operation is further examined to determine if it is an inter-fork (inside a swimlane) or an intra-fork (one that sends a message across swimlanes). The service demand is set to zero for the inter-fork. Moreover, a task entry is generated for each kind of service offered by the corresponding software component instance. The service demands of all the aggregated steps are added together to produce the service demand for the generated phases or activities The above function can also distinguish between LQN activity and phase by checking if there is a fork operation which does not involves a reply. If the condition is true, then the corresponding entry is treated as having LQN activities; otherwise it is transformed to phases.

Figure 9 traces the problem of deciding on activities and phases from UML to IM, and then to LQN. It illustrates how a swimlane of the activity diagram is transformed to the corresponding IM, and then further to LQN. The steps `z1` and `z2` are grouped into `phase1`, while step `z3` becomes `phase2`.

5.2 Generic Transformation Algorithm from IM to a Performance Model

This section introduces a generic algorithm that controls the application of a given set of transformation rules. The purpose is to decouple the transformation control from the details related to the target model. This algorithm can convert a template compliant to a source DTD to another template compliant to a target DTD. The

conversion algorithm takes an input template t_1 and the rule set M and constructs the corresponding output template t_2.

The generic conversion algorithm $T_{(\Sigma, A, G)} \times M \rightarrow T_{(\Sigma, A, G)}$ is given below. The essence of this algorithm is to traverse the input tree and look for a match between different subtrees (i.e., templates) and the left-hand side of a transformation rule from the set M, via the *select-apply* function described in the previous section. If such a match is found in the input template, a gap will be created in the output template, in which the result returned by *select-apply* will be plugged in. The recursive operation *internalconversion* repeats itself for each child vertex.

```
conversion(t1,m){
  t2 = {e}; internalconversion(t1,m,t2);  return t2;
}
internalconversion(t1,m,t2){
  v= first(t1)
  t' = select-apply(v);
  if (t' ≠ null) {
    v'= gaplocation(v);
    g = lab(v);
    gapifyAChild(t2, v', g);
    tplug(t2, g, t');
  }
  if (subtree_t1(v)= {0}) return;
  else{
    S = ti / ti ∈  subtree_t1(v);
    ∀ ti ∈  S
      internalconversion (ti,m,t2);
    return;
  } // end else
} // end internalconversion
```

The above transformation algorithm does not require any specific information on the target template, because the *select-apply* function provides what to plug in, and the *gaplocation* function tells us where to plug it in. More exactly, *select-apply(v)* takes a vertex as parameter and returns one of the following: a null, a vertex or a tree. A null return means that the input vertex has no matching mapping rule. If a vertex is returned, it is a one-to-one mapping. If a tree is returned, it indicates a one-to-n mapping. A null or one-to-n mapping will result in template structure changes.

On the other hand, the *gaplocation(v)* returns a vertex where the newly generated template will be plugged in. It should be noted that both '*select-apply*" and "*gaplocation*" functions are target templates specific, since each target template has its different transformation rules and plug-in rules.

Tow other functions used in the conversion algorithm are *first(t)* and *subtrees$_t$(v)*:

$$subtrees_t(v) = \{select(t, w_i) \mid w_i \in children(v)\} \text{ and}$$
$$children(v) = \{(v_1, v_2, \ldots, v_n) \cdot 1, \ldots, (v_1, v_2, \ldots, v_n) \cdot k\}$$

The former returns the first vertex of a given template, i.e., $v = first(t) = v(1) \in t$. The latter returns all the child templates of v as a set.

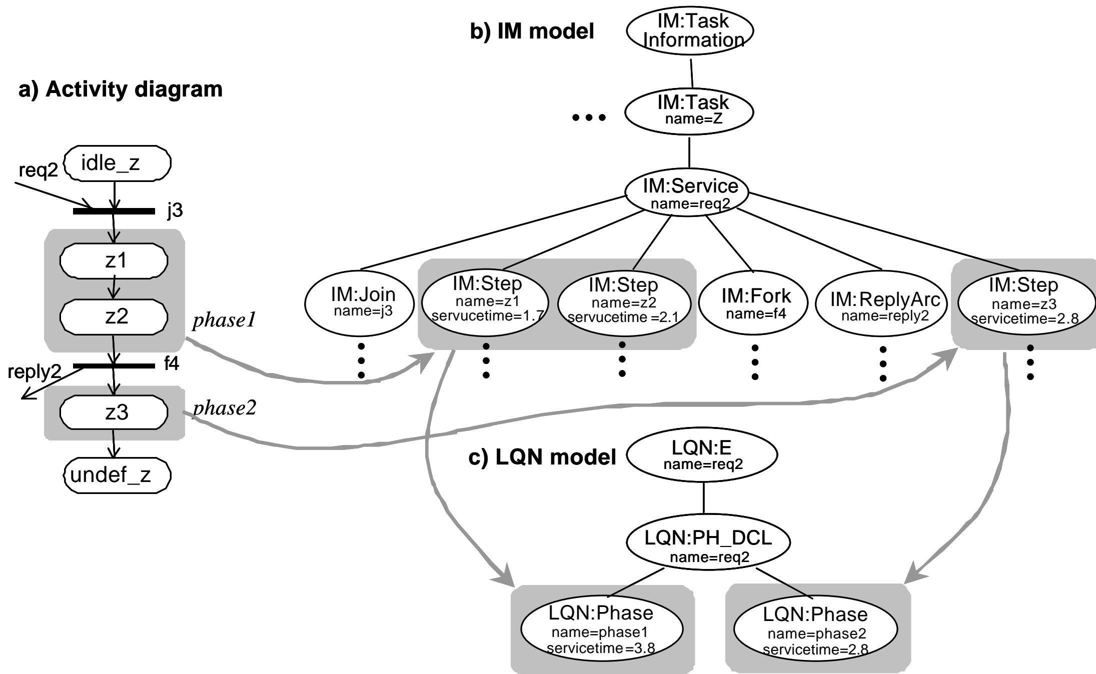


Figure 9. From UML to IM to LQN: aggregating scenario steps into LQN phases

The LQN model given in Figure 10 can be obtained by applying the conversion algorithm described above to the input UML model from Figure 1. This algorithm can be used to transform a IM model to different performance models, provided that a specific rule set M and a *select-apply* function are given for each target model.

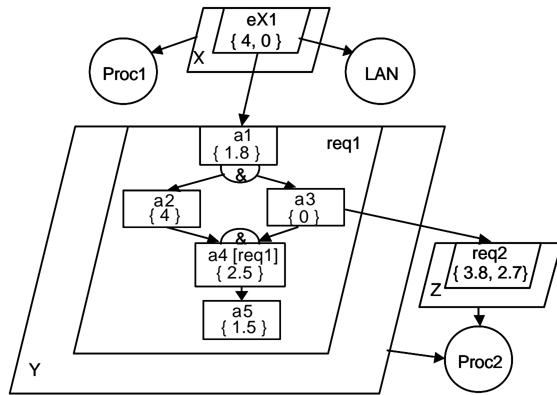


Figure 10. The LQN model for the UML model from Fig.1.

The LQN model contains three tasks (represented by parallelograms), one for each process `X, Y` and `Z`. The processors and network resources are modelled as hardware devices (represented by circles). The tasks `Y` and `Z` are co-allocated on the same processor. Each task has a single entry (drawn as a smaller parallelogram inside the task), according to the grouping of scenario steps illustrated in Figure 7. Entry `req1` of `Y` is composed of activities due to the fact that it contains an internal fork/join, whereas entry `req2` of `Z` is composed of two sequential phases. The LQN model can be used to analyze the performance characteristics of the systems, but this is outside of the scope of the paper, which focuses on proposing a new model transformation method. The use of LQN models for performance analysis of UML designs is discussed in other publications, such as [16].

6. CONCLUSIONS

The main contribution of the paper is a model transformation method that combines concepts from graph transformations with XML transformation techniques, such as XMLgebra. More specifically, the mapping between the input model and the output model is defined at a higher level of abstraction (i.e., at metamodel level) based on graph transformation concepts, whereas the implementation of the transformation rules and algorithm is done at the XML level, using lower-level XML trees manipulations techniques

In order to test the flexibility and modularity of the proposed technique, we have defined transformations from UML 1.4 to two performance target models, LQN and CSIM-based simulation (only the first is discussed in the paper). Current work is under way to update the definition of the transformation rules such that the input models are in UML 2.0. This impacts only the first transformation step UML→IM.

We are in the process of implementing the proposed UML → IM →LQN transformation in XSLT. First are implemented the eXMLgebra tree-manipulation primitives, then the transformation rules and the overall transformation algorithm. In principle, the implementation could be also done in a general-purpose programming language, such as Java. The disadvantage would be that such an approach would require a Java library to read in a UML model from an XMI file and to traverse and manipulate the UML metamodel. Such libraries do exist, but they are specific to different UML tools, and many are proprietary.

The choice of XSLT allows us to take advantage of many existing XML transformation techniques and tools, and avoids the need for a library able to manipulate the UML metamodel. However, XSLT is not as powerful as a general-purpose language, and raises its own challenges. We expect that the experience gained with this method will help us understand better conceptual and practical issues in model transformations. The long-term goal is to build an XML-based model transformation framework that could be used for a large class of model transformations required in the context of MDA.

ACKNOWLEDGEMENTS

This research was supported by Discovery and Strategic grants from the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M., "Model-based performance prediction in software development: a survey" IEEE Transactions on Software Engineering, Vol 30, N.5, pp.295-310, May 2004.

[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] Gordon Gu, D. C. Petriu, "XSLT Transformation from UML Models to LQN Performance Models", Proc. of 3rd Int. Workshop on Software and Performance WOSP'2002, pp.227-234, Rome, Italy, 2002.

[7] Jan Jürjens, Pasha Shabalin, "Automated Verification of UMLsec Models for Security Requirements", Proceedings of UML 2004, Lisbon, Portugal Oct. 11–15,

[8] Christian Kirkegaard, Anders Møller and Michael I. Schwartzbach, "Static Validation of XML Transformations in Java", In IEEE Transactions on Software Engineering, 30(3), pp.181-192, March 2004.

[9] Christian Kirkegaard, "Dynamic XML Processing with Static Validation (Masters Thesis), University of Aarhus, 2003.

[10] J. P. Lopez-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, pp. 25-36, Jan. 2004.

[11] OMG, *UML Profile for Schedulability, Performance, and Time* (SPT), Version 1.0, formal/03-09-01, September 2003.

[12] OMG, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms (QoS)", Adopted Specification, ptc/2004-06-01, June 2004.

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

[14] D.C. Petriu, X. Wang "From UML descriptions of High-Level Software Architectures to LQN Performance Models by Graph Transformations" in Proc. of AGTIVE'99, LNCS 1779, pp. 47-62, Springer, 1999

[15] D.C. Petriu, H.Shen, "Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications", in Computer Performance Evaluation: Modelling Techniques and Tools, (T. Fields, P. Harrison, J. Bradley, U. Harder, Eds.) LNCS 2324, pp.159-177, Springer, 2002.

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

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

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

[19] Schürr, A., "Introduction to PROGRES, an attribute graph grammar based specification language", in Graph-Theoretic Concepts in Computer Science, M. Nagl (ed), LNCS Vol. 411, pp 151-165, Springer, 1990

[20] Schürr, A., Programmed Graph Replacement Systems, in G.Rozenberg (ed): Handbook of Graph Grammars and Computing by Graph Transformations (1997) 479-546.

[21] C.M. Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, Jose Merseguer, " Performance by Unified Model Analysis (PUMA)", accepted at the 5th Workshop on Software and performance WOSP'2005, Palma de Mallorca, Spain, July 11-14, 2005.