Performance Modeling from Software Components

Xiuping Wu, Murray Woodside
Carleton University
Dept. of Systems and Computer Engineering
Ottawa, ON, K1S 5B6
1(613)520-5721
{xpwu | cmw}@sce.carleton.ca

ABSTRACT

When software products are assembled from pre-defined components, performance prediction should be based on the components also. This supports rapid model-building, using previously calibrated sub-models or "performance components", in sync with the construction of the product. The specification of a performance component must be tied closely to the software component specification, but it also includes performance related parameters (describing workload characteristics and demands), and it abstracts the behaviour of the component in various ways (for reasons related to practical factors in performance analysis). A useful set of abstractions and parameters are already defined for layered performance modeling. This work extends them to accommodate software components, using a new XML-based language called Component-Based Modeling Language (CBML). With CBML, compatible components can be inserted into slots provided in a hierarchical component specification based on the UML component model.

General Terms

Performance, Design, Languages

Keywords

Software performance, layered queue model, software component, generative programming, modular submodel, CBML, LQN, performance prediction.

1. INTRODUCTION

In recent years, Component Based Software Engineering (CBSE) has emerged as a promising paradigm for software engineering with interest in both academic and industrial communities [8], [14]. It brings higher efficiency and better quality to software development by using reusable and configurable software components. This also offers some potential advantages for performance engineering [19] [21]. Since performance properties of a software component can be "described" in advance, performance sub-models can be built for each software component and stored in a library for reuse. Very often, when a system is planned, the performance model has to be built from scratch even though it may have many pre-existing components. This is tedious and error-prone work for large and complex systems. If we can re-use the performance sub-models, it should be easier to build the system models. By re-using these sub-models, system models can be built quickly for many different configurations which are tied to software configurations.

As a matter of fact, component-based modeling has appeared as a well-proven approach in some domains, with domain specific library of components. Some examples are Hyperformix and OpNet. In OpNet, the tool uses a library of pre-built performance sub-models for network protocols at different layers, network devices such as routers, switches and workstations and others. The Hyperformix Strategizer tool also uses a set of pre-built performance sub-models for software and hardware. Those sub-models are stored in a library as well. Layered Queuing (LQ) Modeling language [16] [18] is a sort of component-based modeling language which models components by tasks and interfaces by entries. The research work described in this paper extends LQ's modeling capability to include component sub-models.

The goal is component-based modeling which matches the capabilities of component-based software engineering and generative programming [4]. We expect that a component library will be specific to a domain like web services, or to the elements of a single product line [3]. On the other hand, we want the performance sub-models which can be bound flexibly to the system. This is achieved by allowing nested components, and by introducing variable parameters to a component. We also want the specifications of the performance sub-models to be capable of matching the software component specifications, particularly, the UML 2 component model specifications [11] since the UML has now become a standard for software-modeling notation and gained wide acceptance in the industry.

A general approach to developing component based software systems [19] [21] is shown in Figure 1 below.

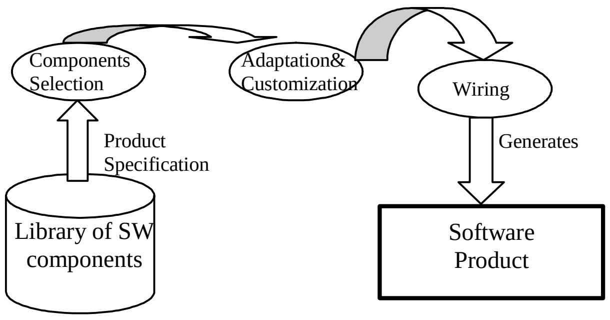


Figure 1 Component Based Software Development

Based on the pre-defined and calibrated performance components and following the same specification for the product, performance models can be created as well. This process is illustrated in Figure 2 below, including a tool LQComposer that we have developed to automate the generation of system models, based on a library of performance components and an assembly model.

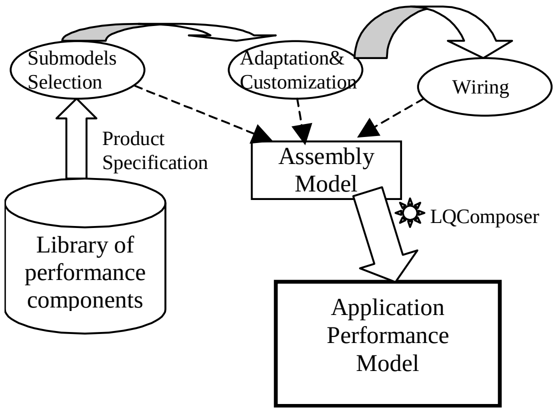

Submodels Selection
Adaptation& Customization
Wiring
Product Specification
Assembly Model
LQComposer
Library of performance components
Application Performance Model

Figure 2 Component-Based Performance Modeling

In previous work the goal of a model-generator that is matched to a software product generator, was described in [19]. In this paper a model builder and a language are described. CBML is an XML (eXtensible Markup Language) [10] based language designed to describe layered queuing models with embedded components, and also the component sub-models. This language will be introduced in the next section. LQComposer creates a solvable LQ model from a system definition with component bindings.

2. COMPONENT-BASED SYSTEM SPECIFICATION

2.1 Specifications for Software Components

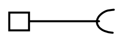
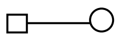
Software components are the basic building blocks in the component based development. However, there is no unified definition for software components. Szyperski and Councill et. al. gave their own definitions in the books [14] [8] respectively. Generally speaking, a software component exhibits the following characteristics. It is an independent, compositional and deployable unit which has clearly defined and documented interfaces interacting with other components. Each component has certain functionalities and may have explicit context dependencies such as operating system or other software components.

We will take our notation for components from the UML 2 proposal [11]. A component is a modular unit with well-defined interfaces; it can be replaced by any unit that has the equivalent functionalities and compatible interfaces. The interfaces of a component are classified as *provided* interfaces and *required* interfaces. Provided interfaces have defined a formal contract of services that the component provides to other components while required interfaces have defined the services that it requires from outside in the system in order to function. These interfaces may optionally be organized through ports. The replacement of a component may take place at either design time or run-time. The substituting component should be able to interact with other components or its environment provided that the constraints of the interfaces are followed.

In UML 2, a component can have two different views, external view and internal view. The external view is also known as a "black-box" view in which it exhibits only the publicly visible properties and operations which are encapsulated in the provided and required interfaces. The wiring between components is specified by dependencies or connectors between component interfaces. The internal view is a sort of "white-box" view which shows the component internals that realize the functionality of the component. An external view is mapped to an internal view by using dependencies which are usually shown on structure diagrams, or by using delegation connectors that connect to the internal parts which are shown on composite structure diagrams.

Some selected notations for describing components in UML 2 are shown in the following table. We use the notation for an interface with a port.

Table 1 Some Notations in UML 2 Components Diagram

Graphical Notation	Description
<<component>> AppServer	AppServer is a component. Optionally, it may also have two protruding rectangles in the upper right-hand corner.
	A required interface of a component, associated with a port.
	A provided interface of a component, associated with a port.

In our graphical notation for CBML we will later customize the UML notation to a square on the boundary to show a required interface, and a circle to show a provided interface.

2.2 A Management Information System

This section presents some component diagrams in UML 2 for a Management Information System (MIS) that we have studied. These diagrams will serve as examples that will be compared with the performance component models that we have proposed. The performance component models will be introduced in the next section. The comparison will show how closely they are matched.

This MIS is a typical four-tiered E-business system which consists of client browser, web server, application server and database servers. Clients send requests to the web server which does some processing, and then invokes the application server which does the specific business computing and invokes operations on the database servers backend. After the results have been worked out, they are sent back to the clients. There are two types of requests that clients send to the system, reporting requests and viewing requests.

The structure diagram in Figure 3 below shows the composite structure of components. The wiring between components is represented by assembly connectors between provided and required interfaces.

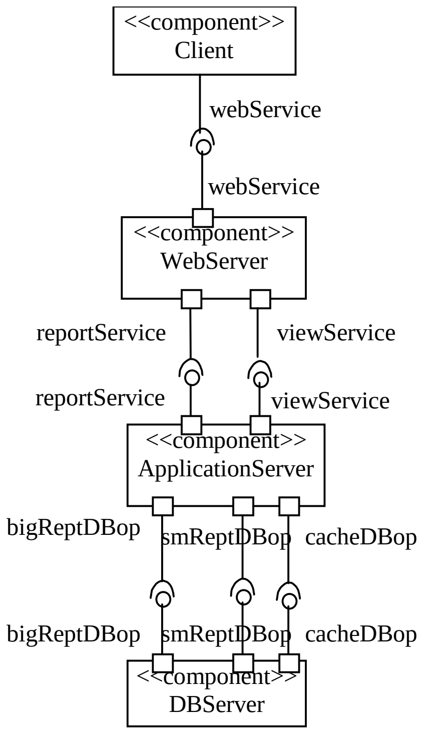


Figure 3 A Composite Structure of Software Components in the Management Information Systems, in UML2

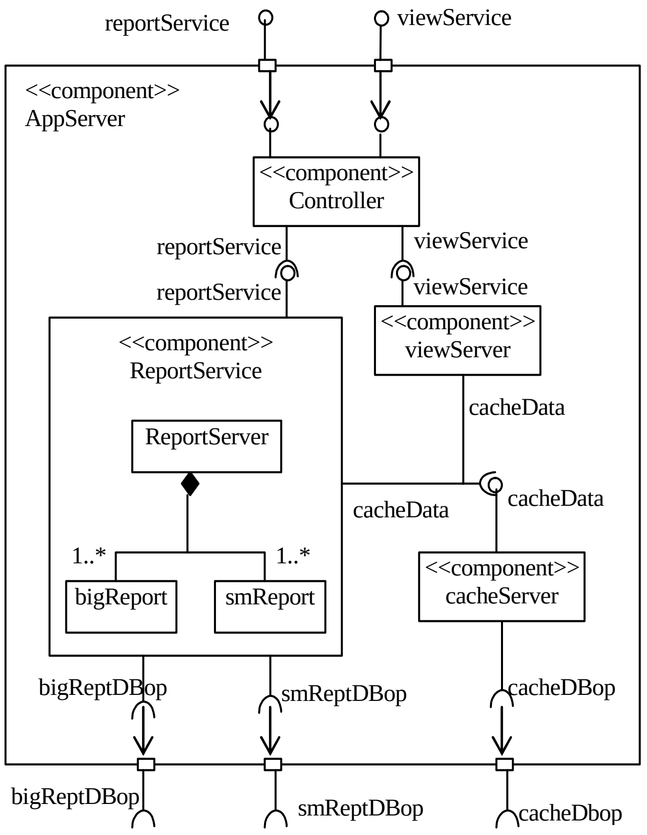


Figure 4 An Internal View of the Software Component AppServer, to be plugged into ApplicationServer

Focusing on the Application Server in Figure 3, we see its external view in the middle of the Figure. The internal view of a design for this component, with nested components inside it, has been renamed AppServer in Figure 4. The reasons for this will be clearer below. The nested components are shown with bindings between provided and required interfaces, described by dependencies.

The controller interprets the service requests coming to the application node. Based on the type of the requests, reporting service or viewing service will be invoked. These two services all need to access the cache server which provides caching data for reuse. The reporting service and viewing service all need some database operations. The report server can generate two different kinds of reports, big report and small report namely. These different reports require very different database operations.

When we come back to this example we will use a modified notation which provides more flexibility.

3. LAYERED MODELING OF COMPONENTS

3.1 Layered Queueing Network Models

The Layered Queuing Network Models (LQN) [16] extends the traditional queuing network models in that they are capable of capturing the impact of multiple layers of software servers. The structure of an LQN model resembles the software architecture of a system. An LQN model is expressed as a set of objects called "tasks". A task provides services which are represented by "entries". Entries of one task make calls to entries of others at lower layers. Tasks are executed on processors. The LQN modeling language implies the component-based modeling in that a task actually represents a component while an entry represents an interface, or part of one (a subset of the interactions at the interface, grouped together for their performance characteristics), thus one interface may be modeled by several entries, or may be split up before modeling by entries. Therefore, a task is a kind of component which has provided services represented by entries and required services represented by calls to other entries. A task has to be executed on a processor which is a required service, too. The modeling of sequential executions, parallel executions (AND Forks and Joins), alternative executions (OR Forks and Joins) as well as repetitive executions is accommodated by "activities". Activities are the smallest unit of computation. It can accept requests from entries or send requests to entries. The details of modeling with activities can be found in paper [7]. In LQN models, both nested components, and components giving the required services, are in lower layers (layers are a control hierarchy rather than a structure hierarchy).

An LQN model includes the following workload parameters that are used to describe software:

1. Call parameters. This involves the call patterns between a caller and an invoked component or method in the software system, and the average number of calls. The call patterns are classified as synchronous, asynchronous and forwarding calls which are tied very closely to the software.

2. Host demands. This refers to the total average amount of time that is consumed by the operations on that host in order to complete a service that is represented by an entry. More details of the operations, showing the sequence of operations, are represented by activities.
3. Scheduling policy such as FIFO (First Come First Serve), PS (Processor Sharing) and priority based. This is exactly the same as appeared in the software.

Overall, LQN models are suitable for modeling component-based systems. This research has extended LQN models to support sub-models, sub-model libraries and sub-model compositions.

3.2 Components in Layered Models

An LQN component is a pre-constructed performance sub-model that captures the performance attributes of a software component or a subsystem. These components are termed CBML components which are associated with the CBML language that we have proposed and designed.

We wish to be able to plug different components into a compatible location in a system, so the internal view (still called a "component") is separated from the external view, which will be called a "slot". A CBML component can be plugged into a compatible slot. It may also have parameters that can include CPU demands, service request parameters and configuration parameters such as threading levels. Therefore, a CBML performance component is a parameterized sub-model that captures the performance attributes of a software component. Its parameterization reflects variations in design, in features and in deployment and configuration. A performance component can be instantiated one or more times within a model, with different parameters each time.

A CBML component has interfaces which are classified as *incoming* interfaces or *in-ports* and *outgoing* interfaces or *out-ports* that correspond to the *provided* interfaces and *required* interfaces as specified in UML 2 for software components.

An interface of the CBML component may define a type associated with it.

The deployment of the component to the actual physical processors is defined by *re-bindable* processors which represents the concept of "configuration interface" that Bosch has proposed in his book [2].

3.3 External View of CBML Component

The external view of a CBML component is by means of a slot with interfaces that connect the performance component into the system model. At this point the in-ports are represented by circles, and the out-ports by squares on the slot boundary. These may be seen as customizations of the UML2 component notation, as described above.

An external view of a performance component that corresponds to the software component AppServer shown in Figure 4 is illustrated in Figure 5 below.

In this view, <<slot>>, <<in-port>>, <<out-port>> and <<component>> are introduced as performance stereotypes. The concept of slot is introduced as a placeholder for a performance component that has compatible interfaces. The slot specifies how a compatible component can be plugged into the system model

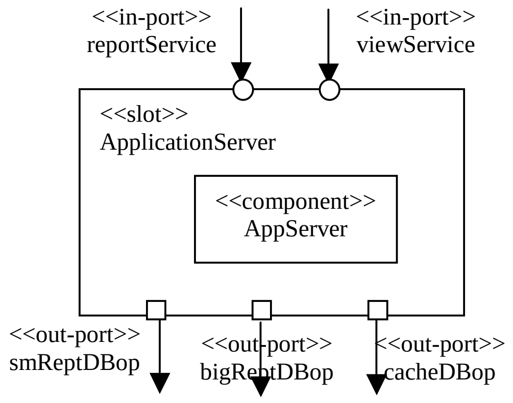


Figure 5 An External View of the Performance Component AppServer in the MIS

and connected to the rest of the model. The specification of wiring is through binding that is defined within the slot.

A slot has <<in-port>> which is an incoming interface and will be connected to the <<in-port>> of the bound component. Its <<out-port>> is an outgoing interface and will be connected to the <<out-port>> of the bound component.

The interfaces of the slot fill in the role of a black view for a compatible performance component. The lollipop sign represents an in-port and the square sign means an out-port.

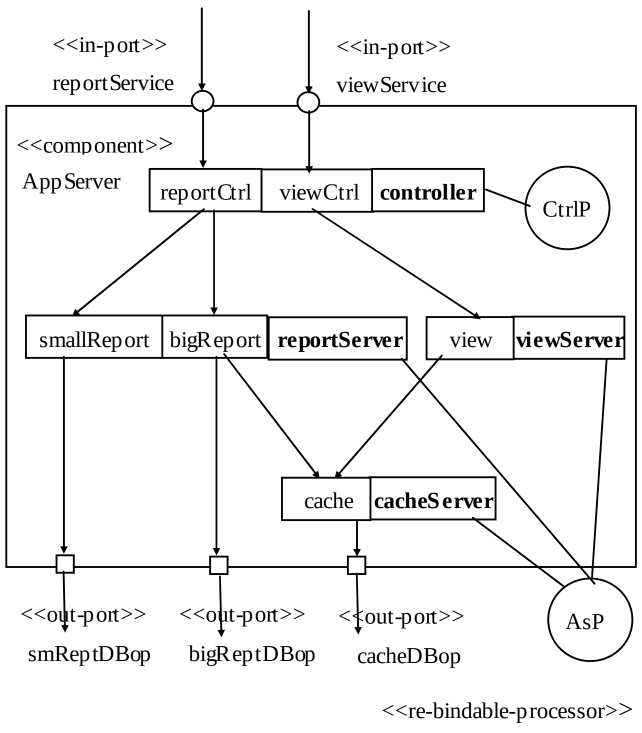


Figure 6 An Internal View of the Performance Component AppServer in the MIS

3.4 Internal View of CBML Component

The graphical view of the performance component AppServer is illustrated in the Figure 6.

The <<in-port>> represents an *incoming* interface of the performance component which corresponds to the *provided* interface of the software component in UML 2. The <<out-port>> represents an *outgoing* interface that corresponds to the *required* interfaces of the software component.

An <<in-port>> represents the services that the component (sub-model) provides to its clients while an <<out-port>> means the services that the component must obtain from its outside in order to function. The <<re-bindable-processor>> represents the processor that can optionally be replaced by a system processor. If it is not replaced, the processor defined in the component will be instantiated in the model.

In this diagram, the internals of the component model are denoted by some notations used in LQN models. The rectangles that have bold letters inside represent tasks which are a reduced form of performance components. The rectangles at the left side of those are entries which are equivalent to the provided interfaces of the component. The arrows coming from these entries represent requests (calls) that the entry makes to other entries of other tasks at lower layers. These requests are equivalent to the required interfaces of a component. The big circles inside and outside of the components represent the processors that the tasks are running on. Processors are a kind of service that is required by the tasks which represent a component in a reduced form.

In this component model, the big circle inside the component represents an internal processor which is not visible to the outside and can not be replaced by other system processors. The outside processor denotes a replaceable processor that may be replaced by a system processor in the system model. If no system processor is specified at instantiation time, then it will be instantiated as the default one for the component.

To describe a component model that will be stored in a library for reuse, the following details are needed.

- Interfaces: incoming and outgoing interfaces which are defined as in-ports and out-ports in CBML language. These interfaces are mapped to the provided and required interfaces of the software component.
- Re-bindable processors: processors that can be replaced by system processors or other component processors in which this component has been deployed.
- Workload parameters: CPU demands and service requests parameters such as the mean number of invocations of a service of a component. There maybe variables that reflect the performance attributes of the software component in different context. There are several approaches to obtaining these parameters which can be found in [12] [13].
- Configuration parameters: threading levels and number of replications. These can also be variables that can be instantiated with different values in different situations.

3.5 A Nested Component

A place for a nested CBML component can be defined by a slot within the outer component. As for the outer component, the binding of a component in this slot is indicated in its binding section. The required services must be bound to entities in the immediate outer context, that is in the higher level component. Instantiation parameters of the inner component can be passed through the outer component, to be set in the outermost model.

The AppServer component shown in Figure 6 has a task ReportServer with two entries. To replace it by a subsystem we define the slot rptS in its place, as shown in Figure 7. The slot has input and output interfaces for the entries and calls of the task. The rebindable processor rptP is bound to another rebindable processor AsP in the outer component AppServer, which will eventually be bound to a system processor

We will exploit this nested slot by attaching a replication parameter to it, so that the inner subsystem is replicated for higher performance.

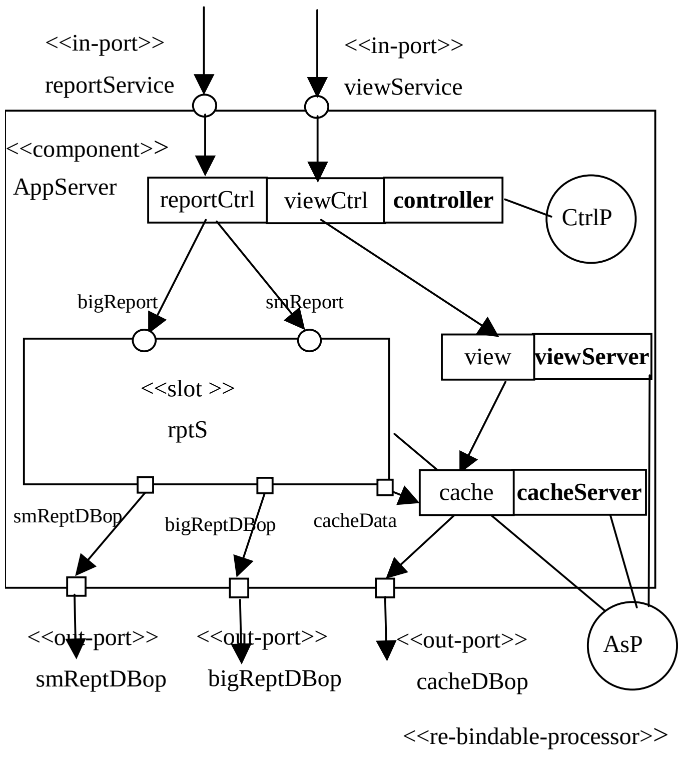


Figure 7 Component AppServer with a Nested Component ReportServer inside

One inner component that could be bound into the slot is shown in Figure 8, with its bindings. The smallReport and bigReport functions are provided by separate servers. Both are executed on a single processor rptP, which is made rebindable, and is bound here to the same processor as before (that is, to AsP). A more complex subsystem model could be defined and bound into the slot instead.

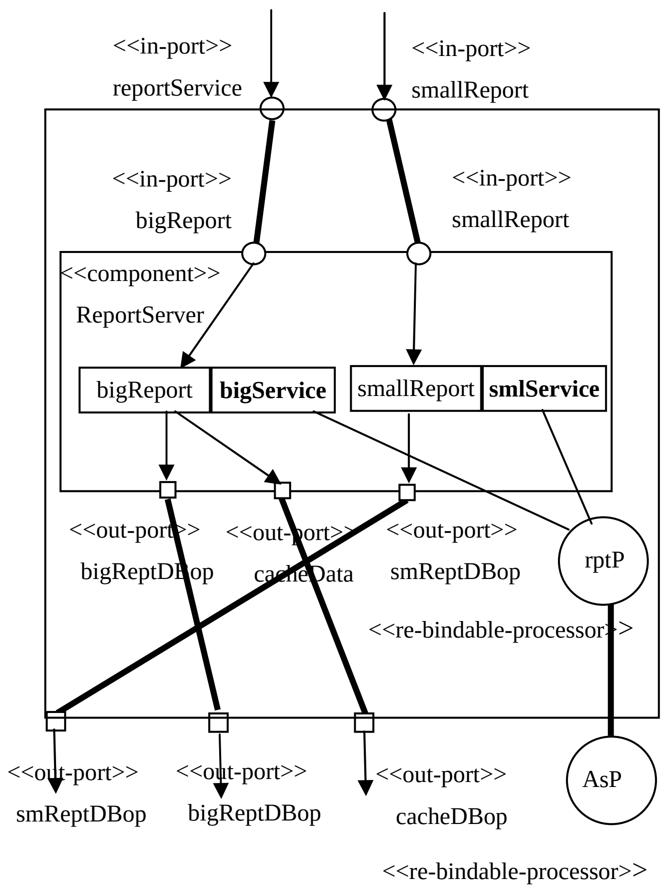


Figure 8 An Internal View of the rptS slot, with the ReportServer component bound to it

The number of the ReportServer that need to be instantiated in the model can be specified by a parameter in the binding section of slot, as shown in the next section. Due to the space limitation, we will only give one example that shows two replicated report servers instantiated in the AppServer component model. This example is shown in Figure 9.

Figure 9 shows the AppServer component definition produced when the inner component is bound in place. The interfaces of the slot and the ReportServer have now disappeared after the instantiation, and the calls have been redirected to the appropriate targets. The tasks inside the dashed box are now running on the same processor AsP which is a replaceable processor of the component AppServer.

4. THE CHALLENGE OF PERFORMANCE CHARACTERIZATION OF SOFTWARE COMPONENTS

The approach taken here to creating slots and components essentially applies the full power of the modeling language for systems, to component subsystems. It follows that when the component is bound into the slot, the resulting complete model is not constrained by the fact that components were used in defining it. The result should be that the performance effect of components can be fully characterized. We make this claim, and it is useful to examine it in the context of differences between components. Thus, we will consider how variations between components are captured.

In software design, the use of components and other generative elements provides an opportunity to reuse software and also to insert variations into a system [4]. Variations can arise both due to changes in features, and due to design differences that provide the same features (to clean up an aging architecture or to meet a new challenge such as larger scales of deployment, for example). When describing the performance models for the same components, there is a challenge of representing the performance effects of these different kinds of variations. We call this the performance characterization of the variations.

In this work the difference between two components that can fit into the same slot in the overall system can be represented, in the performance model, by:

- a difference in the CPU demand of an operation, within a constant structure of tasks and entries. This might represent a change in the features offered by the product, or in the algorithms or data structures, without any change in concurrency or partitioning of responsibilities.
- use of different nested components, which could also represent a change to the product features, algorithms or data structures.
- a difference in numbers of threads of tasks, or in the software configuration in its environment (such as the allocation of tasks to processors, or the allocation of priorities to tasks). This affects performance through concurrency and competition for resources. One may argue that some of these are not properties of the software component itself, but of its deployment, but in any case these deployment differences can be defined.
- a different pattern or amount of demands for services between entities within the submodel, or for required services, representing a different partitioning of responsibilities within the component subsystem, or a different pattern of communication between entities (for example the use of call-backs).
- different required services, due to changes in features offered, or in the partitioning of responsibilities between subsystems.
- a different architecture for the subsystem (a different set of internal tasks and entries, with different service relationships and concurrency). This might represent a different control structure, or large-scale design approach, for example using pipelining versus a master-slave architecture. It might also arise from adapting the software to a different system infrastructure, for example adapting a design that uses CORBA for service access, to use Enterprise Java Beans, or a Web Services approach instead.

The above list captures essentially all the software variations the authors have been able to gather together. It also suggests how models for software features and their variations may be represented in a library.

An important case is a component subsystem which is simply a set of sequential objects intended to run in a single process. Variations with different algorithms and data structures differ only in their CPU demands and possibly their input/output demands. They have no differences at the level of cooperating tasks. Although the component is intended to be just part of a process, it can be modeled as a task (really a pseudo task, since it is not really concurrent) with entries representing the major methods, called by the process that invokes the object. The calls should be blocking, since calls in sequential processes are blocking; in this way its execution is included in that of its caller. It should have infinite threads since it does not form a task queue, and its execution must be allocated to the same processor as its caller so the processor workload is correct. If it is included in several processes on different processors then it must be instantiated separately for each processor.

In this simple example, the different variations can all be represented by one component definition with different sets of parameters. Any one of the variations can be constructed, by substituting parameter values. This suggests the possibility of generating a wide range of component submodels from compact parameterized representations. The language CBML defined below provides support for the parameters.

The example also shows (in the last sentence) that a component may have to specify some constraints as to how it can be bound. Other constraints have to do with correct use of a component, by matching the type of a request with the type of an offered interface. For that purpose it is necessary to type the interfaces in slots and component submodels, and to require some form of matching. In general there is an issue of compatibility of a component with a slot, which is addressed by the proposed scheme but which will not be described in detail here. Compatibility is addressed partly by typed interfaces; other aspects include compatibility of components with the processor platform, and the network protocols that some components use.

5. THE CBML LANGUAGE

In this work, the LQN language that describes LQN models has been extended in order to have more capability and flexibility to model software components and component-based systems. This extended language is based on XML language and termed CBML (Component-Based Modeling Language). A CBML model is organized as a structured model which is suitable for XML language to describe. The XML schema (XSD schema) describes the allowable contents and datatypes that a CBML document may have. Some concepts in this modeling language are illustrated in the following figures. These figures were generated by the XMLSpy schema editor.

5.1 The Core of CBML

The core elements that are defined in an LQN model are shown in Figure 10. The element slot has been introduced to support nested component definition as mentioned in the previous sections. In UML 2, the nested components are defined through bindings between provided and required interfaces. In our CBML performance components, the wiring is done through slots and bindings which are defined within the slot. The details of the slot will be introduced later.

A task now has a service point associated with it. This service point acts as a required interface of the task. Therefore, a software component can be modeled by a task in a reduced form in which, its provided interfaces are represented by the task's entries while the required interfaces are represented by the task's services.

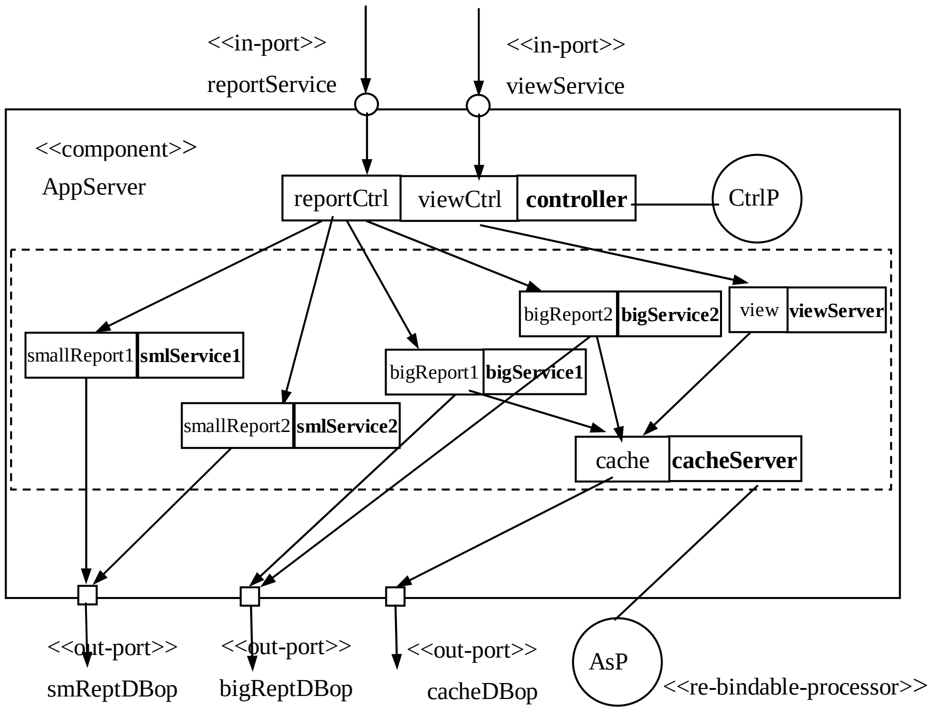


Figure 9 An Internal View of AppServer with two replicated ReportServers components expanded inside it. Processor AsP is bound to all the tasks within the dashed line.

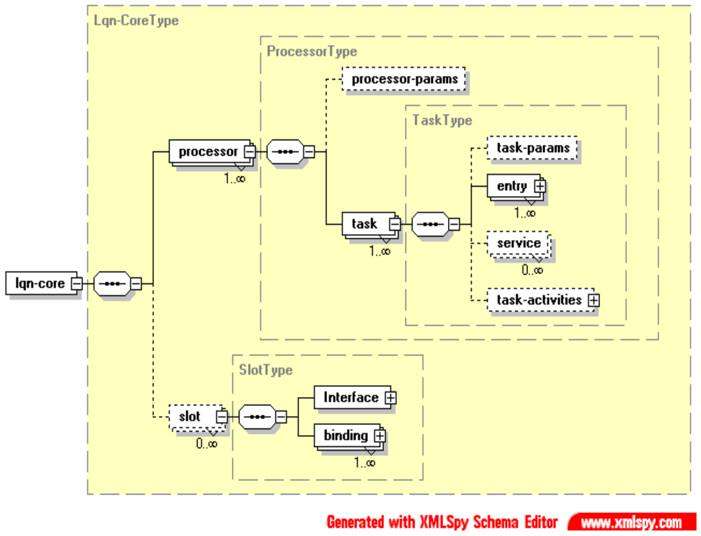


Figure 10 Elements defined in LQN Core

However, the details of a software component can be modeled by the performance sub-model which has explicit interfaces which correspond to the software component design specifications as described by UML 2. This sub-model will be introduced in the next section.

In the diagrams that were generated by XMLSpy Schema editor, the plus sign means that the details of the element have not been expanded. The symbols such as 0..∞ and 1.. ∞ represent the cardinality of the element that will appear in the model. The dashed lines mean optional elements while solid lines represent required elements in the model definition. The switch signs represent alternatives. The attributes of the elements have not been shown in these diagrams.

5.2 The CBML Sub-model Definition

The elements that are defined in CBML Sub-model are illustrated in the following Figure 11. Again, the attributes of the elements have not been shown.

A CBML sub-model has defined a section of Interface which includes in-port, out-port and Replaceable-Processors that have been introduced in the previous section. The attributes of the in-port and out-port record the information associated with them, such as the name of the port, any description of the port and the source or target that they are connected from or connected to inside the component.

The element Parameter defined in the sub-model is used for specifying component variable parameters. The variable name starts with a ‘$’ sign. This element has also defined a default value for the parameters. When the component is instantiated in a system model, if no instantiation value specified for a variable that appears in the component model, then the default one will be used.

A sub-model may or may not have any slots that will bind nested components into this component. The definition of slot will be detailed in the next section.

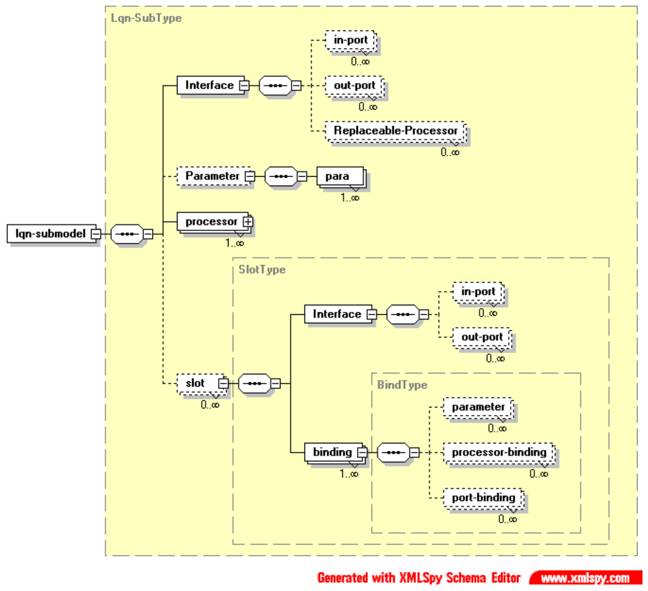


Figure 11 Elements Defined in CBML Sub-model

5.3 The Slot in CBML

The schema definition of a slot in CBML is illustrated in Figure 12 below.

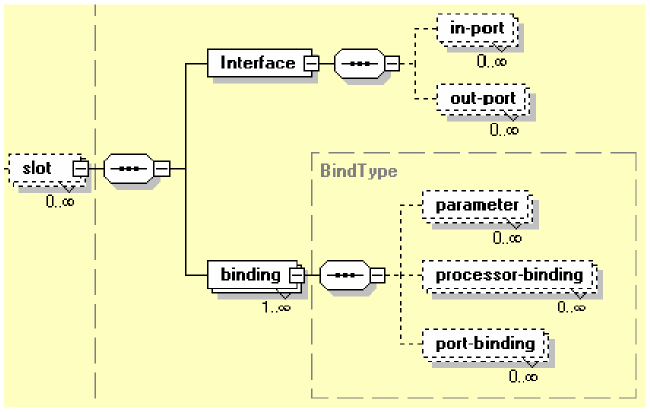


Figure 12 Elements Defined in a Slot

A slot has two main parts: Interface and binding. Interface specifies the in-ports and out-ports that it has. Interface may be associated with a type. The ports have specified the provided services (by in-ports) and required services (by out-ports).

The element binding specifies how a compatible component can be instantiated and plugged into the system model and connected to the rest of the model through this slot. The element parameter is used to customize the component model with appropriate instantiation parameters passed in. The element processor-binding specifies how a replaceable processor in the component model can be replaced by a system processor. This reflects the aspect of component configuration and deployment. The element port-binding specifies how the slot is wired to the component by their ports (interfaces).

The XML schema for the element binding is listed below. An introduction to XML schema can be found in [20].

```
<xsd:element name="binding" maxOccurs="unbounded"/>
  <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="parameter" minOccurs="0"
                         maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="name" type="xsd:string"
            use="required"/>
              <xsd:attribute name="value" type="xsd:string"
              use="required"/>
           </xsd:complexType>
         </xsd:element>
         <xsd:element name="processor-binding" minOccurs="0"
                           maxOccurs="unbounded">
              <xsd:complexType>
                 <xsd:attribute name="source" type="xsd:string"
                          use="required"/>
                 <xsd:attribute name="target" type="xsd:string"
                          use="required"/>
              </xsd:complexType>
          </xsd:element>
          <xsd:element name="port-binding" minOccurs="0"
                            maxOccurs="unbounded">
              <xsd:complexType>
                 <xsd:attribute name="source" type="xsd:string"
                     use="required"/>
                  <xsd:attribute name="target" type="xsd:string"
                     use="required"/>
               </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
  </xsd:element>
```

Using a slot, a single class of component model can be instantiated as several identically replicated component models in the system model if necessary. There is an attribute “replic_num” associated with the definition of slot which can be used to specify how many replicas of the component model are needed. Based on this, the calls coming into this slot will be split equally to the replicated components. This is very useful in the case of modeling several identical software components or sub-systems. An example has been described in section 3.5 where the inner component has two replicas. The next section will provide a practical example of applying this language to model a large commercial software product where nested components and replications are involved.

On the other hand, using slots facilitates the naming issues for the instances of the performance component. Each slot must have an id associated with it which is used as the prefix of the elements in the component model.

5.4 Special Types of Components

A performance study may also require component sub-models that are not part of the product, but represent parts of its environment, such as middleware, file systems, databases or web services used by the product. This class of components was discussed in [17] under the name of “completions”, because they are needed to complete the performance model. In that work it was envisaged that they might be added automatically, based on special annotations and rules. The slot feature described here is suitable for representing explicit services such as databases or file systems. However middleware or network sub-models might be introduced wherever their use is implied by the configuration (e.g., by knowledge of the network used between two specified processors). This would require suitable rules for introducing slots where needed.

6. AN APPLICATION OF COMPONENTS

The component model and language were applied to a software product line in management information systems, described by the overall model in Figure 3. For the study, the slots other than ApplicationServer were represented by LQN tasks, as shown in Figure 13.

Some of the possible variations in ApplicationServer have components with different structure and some have different parameters. Here we will consider parameter changes on the one component AppServer defined in Figures 7 and 8.

There are many possible variations in ApplicationServer. For instance, there may be no caching involved. Or it may need to access some additional databases. Other possible options may be that it needs to access additional servers for specific analysis (e.g. for data mining or optimization). By taking different parameters, the performance component of AppServer can also model the application node whose internals may have different software but still accomplish the similar functionalities with different performance attributes. A typical case of this is that there may be several replicated application nodes deployed on several single processors. There may be only one application node deployed on one more powerful machine (e.g. multiple processors) too. The size of the thread pool for the report server and cache server can also be varied. All these variations can be expressed in the CBML language.

The following types of variations were considered here:

- The variable threading levels can be described by defining the multiplicity of the tasks in LQN model as a variable. When the component AppServer is instantiated, a desired number of threads can then be passed into through the slot which binds the AppServer to the system model.
- The variable number of replications of AppServer can be achieved by giving a specific value for the “replic_num” which is the attribute of slot that is bound to AppServer as described in section 4.3.
- The number of report servers that are running inside the AppServer can also be varied and their replications can also be achieved by specifying the value of the attribute “replic_num” for the slot that is bound to ReportServer in

the component AppServer. This value can be passed from the outside component AppServer.

- The rebindable processor in AppServer can be used without rebinding (to give a separate platform) or bound to a system processor shared with other functions in the system.

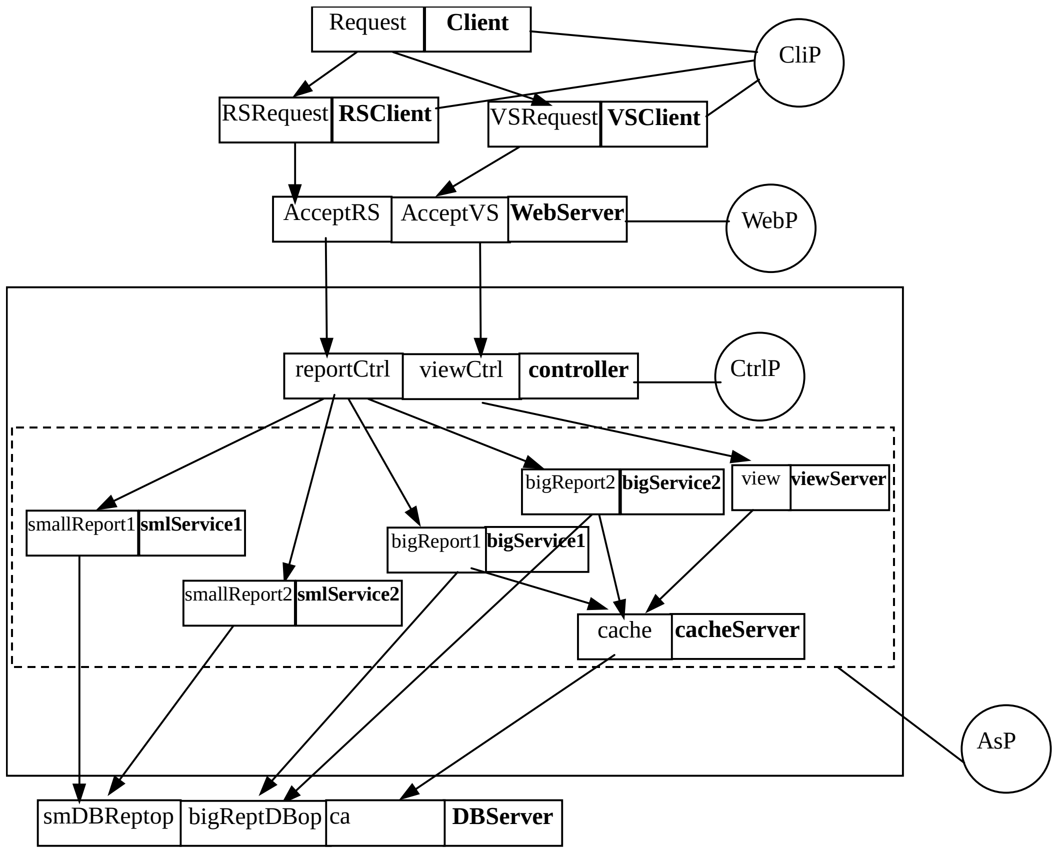


Figure 13 The Final System Model

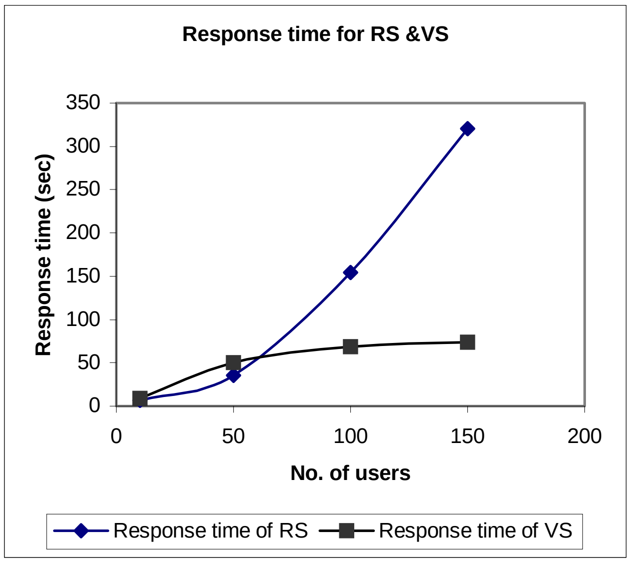


Figure 14 Response Time for the Base Case

6.1 Results for some cases

Some cases were considered that embody these variations, with the results shown below. Performance calculations were done by the LQN solver [5] [6]. Results are given for response times of the Reporting Service (labeled RS) and Viewing Service (labeled VS). As the results are only shown to illustrate the type of the experiments that are possible, the parameters of the model are not given here. The base case has two report servers (each has one thread for the smallReport server and five threads for the bigReport server) and the cache server is single threaded. The application node has two processors. The results of the base case are shown in Figure 14. We can see that the response time for 100 users are about (150, 60) for (RS,VS), respectively.

The next case introduced 10 threads for the cache server and 2 threads for the smallReport server. The results are shown in Figure 15. The response times are both much smaller.

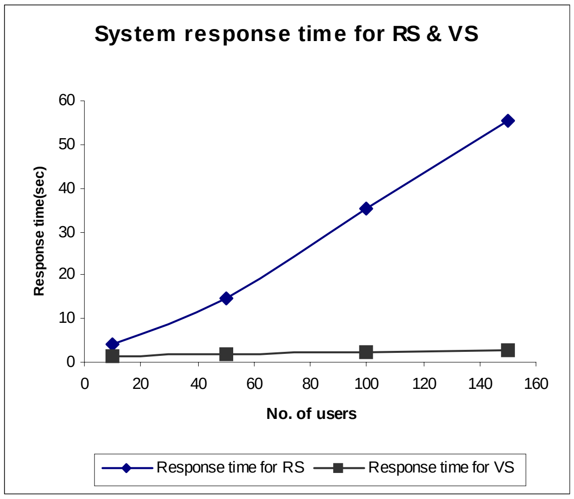


Figure 15 Response Time for Multithreaded Cache Server and SmallReport Server

The following two cases have introduced two replicated AppServers running on two nodes, each with its own single processor.

Figure 16 shows the results for a single threaded Cache Server. The response times are better than the base case with no replication, but worse than Figure 15, which includes Cache Server threads.

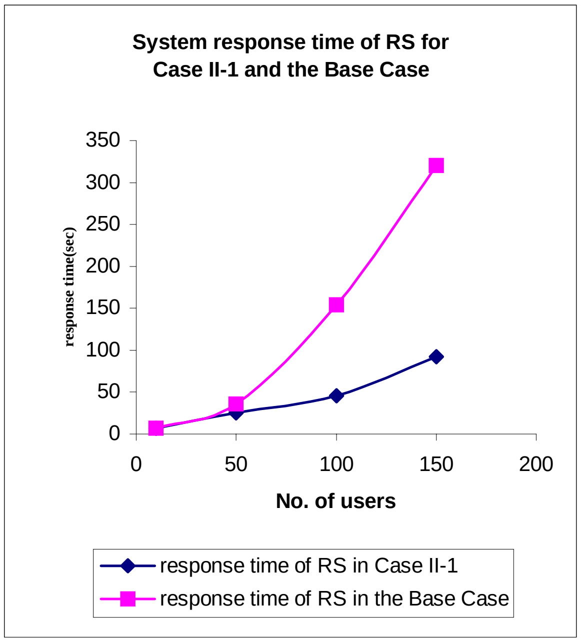


Figure 16 Response Time for Two Replicated AppServers and Single-Threaded Cache Server

Figure 17 shows results with two replicated AppServers and a Cache Server with 10 threads. The response times for 100 users are now (RS, VS) (10, 8) roughly, which is the best so far.

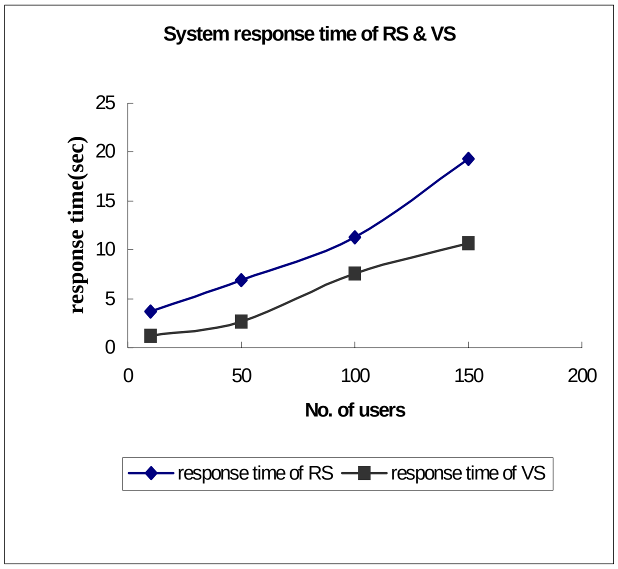


Figure 17 Response Time for two Replicated AppServers and 10 threads of

The example results show how certain variations, which were of interest in an industrial context, could be addressed using components and parameters. The results indicate that the cache server really must be multi-threaded, and that this is more effective than additional report servers.

The cases described here only don't exploit all the features of the model and the language. Other examples have exploited the ability to bind different submodels, for instance.

7. RELATED WORK

Hissam et.al. have argued in favour of a "prediction-enabled component technology" (PECT) [9] which includes analysis for prediction of the assembly-level properties of a composed system. Their concept of analysis is very general, but this work would fit well into PECT. They provide an example of performance analysis, which is much simpler than the systems described here (for example, it does not include concurrency or contention).

Bertolino and Mirandola [1] describe a CBSE performance framework, in which composition is defined by UML sequence diagrams for the interactions between components. They do not create submodels of components, and they cannot replace one component by another, without going through all the modeling steps again. Components are modeled only at the object level, with no mention of concurrency within a component.

Languages for software composition are also related to CBML. One such language, also based on XML, is XCompose [15]. As with CBML, XCompose specifies the services offered by the component (its methods) and the services it requires from its environment. Unlike CBML however, it specifies composition in terms of a usage scenario or program, in which the methods of the component are called. CBML describes the number of service calls to a component, rather than the sequence; this is a more aggregated description suitable for many performance calculations. In fact it is also possible to describe sequences of calls to component interfaces in CBML. Thus the similarity in language facilities is quite strong.

8. CONCLUSIONS

The language CBML (based on XML and UML2) describes performance models of software components and component-based systems. It has the capability to capture the performance-related features of software components, their integration and deployment in the system, and variations between alternative components in a product line. It has been applied to an industrial software product, and met (or exceeded) all the needs of the study; some of the results have been briefly described.

A model assembler tool, not described here, generates performance models automatically using a library of component sub-models. The performance components are reusable, just like the software component themselves. The language and the tool have been applied to model a commercial software product in a software product line.

The combination of layered performance modeling and components described at an architectural level, as in CBML, appears to be well suited to the analysis of software product lines and other kinds of component-based systems. Some language features that will be required to specify constraints on components have been indicated but are not described in this paper.

9. ACKNOWLEDGMENTS

We gratefully acknowledge financial support provided by Nortel Networks through a postgraduate scholarship, and by NSERC (the Natural Sciences and Engineering Research Council of Canada). Discussions with Godfrey Lee, Jim Miller-Cushon and Bob Minns helped us to understand the case study application.

10. REFERENCES

[1] A. Bertolino and R. Mirandola, "Towards Component-based Software Performance Engineering," in Sixth ICSE Workshop on Component-based Software Engineering, Portland, Oregon, May 2003, pp. 1 – 6

[2] .J. Bosch, “Design and use of software architectures; Adopting and evolving a product-line approach”, Addison-Wesley, 2000

[3] P. Clements and L. Northrop. “Software Product Lines; Practices and Patterns”, Addison-Wesley, 2000

[4] Czarnecki, K. and U. Eisenecker, “Generative Programming”, Addison Wesley, 2000.

[5] G. Franks, A. Hubbard, S. Majumdar, J. Neilson, D.C. Petriu, J.A. Rolia and C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", Performance Evaluation, vol. 24, pp117-136, 1995

[6] R.G. Franks, S. Majumdar, J.E. Neilson, D.C. Petriu, J.A. Rolia and C.M. Woodside, "Performance Analysis of Distributed Server Systems", *Proc. Sixth Int. Conf. on Software Quality*, Ottawa, Oct. 28-30, 1996, pp. 15-26.

[7] Greg Franks, Murray Woodside, "Performance of Multi-level Client-Server Systems with Parallel Service Operations", Proc. First Int. Workshop on Software and Performance (WOSP98), pp. 120-130, Santa Fe, October 1998

[8] G. T. Heineman and W. T. Councill, “Component-Based Software Engineering; Putting the Pieces Together ”, Addison-Wesley, 2001

[9] S. A. Hissam, G. A. Moreno, J. A. Stafford, and K. C. Wallnau, "Packaging Predictable Assembly," in Component Deployment 2002 (CD2002), Berlin, June 2002.

[10] M. Kempa and V. Linnemann, “XML-Based Applications Using XML Schema”, XML-Based Data Management and Multimedia Engineering-EDBT 2002 Workshops, Springer, Lecture Notes in Computer Science.P67-P90

[11] Object Management Group, “UML 2”, ptc/03-08-02, August 2, 2003 (Accessible from http://www.omg.org/docs/ptc/03-08-02.pdf)

[12] D. Petriu, M. Woodside, "Analysing Software Requirements Specifications for Performance", Proc. Third Int. Workshop on Software and Performance, Rome, July 2002

[13] K. H. Siddiqui and C.M. Woodside “Performance aware software development (PASD) using resource demand budgets” In the Proc. of the Third Int. workshop on Software and Performance, pp.275 – 285, July 2003

[14] C. Szyperski, “Component Software; Beyond Object-Oriented Programming”, Addison-Wesley, 1998

[15] N. Tansalarak, K.T. Claypool, "XCompose: An XML-based Component Composition Framework", Third Int. Workshop on Composition Languages, 17th European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany July 22, 2003.

[16] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", IEEE Transactions on Computers, Vol. 44, No. 1, January 1995, pp. 20-34

[17] M. Woodside, D.B. Petriu, K. H. Siddiqui, "Performance-related Completions for Software Specifications", Proc 24th Int. Conf. on Software Engineering (ICSE 2002), Orlando. May 2002.

[18] M. Woodside, “Tutorial Introduction to Layered Modeling of Software Performance”, Edition 3.0, May 2002 (Accessible from http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialg.pdf)

[19] Xiuping Wu, David McMullan, Murray Woodside, “Component Based Performance Prediction”, 6th Int. Workshop on Component-based Software Engineering, part of ICSE 2003, Portland Oregon, May 2003.

[20] R. A. Wyke and A. Watt. “XML Schema Essentials”, Wiley Computer Publishing, 2002

[21] S. Yacoub. “Performance Analysis of Component-Based Applications”, Proceedings of the Second Software Product Line Conference, pp.299-315, San Diego, CA, USA, August 2002