Towards a Framework for Reasoning about the Performance of Component Software

Ksze Kae Ker and Jun Han
School of Information Technology
Swinburne University of Technology
John Street, Hawthorn, VIC 3122, Australia
E-mail: kker@it.swin.edu.au jhan@it.swin.edu.au

Abstract

The issues concerning the performance of commercial-off-the-shelf (COTS) components have gained much attention in the research community recently. Predicting the performance of a system constructed from components is beneficial for reasons such as bottleneck identification and performance tuning. However, the research status in this area is still relatively immature. Component technologies such as .NET and Enterprise JavaBeans offer assembling platforms but do not provide any insights into the system performance before construction.

In this paper, the Performance Characterisation And Composition (P-CAC) framework is used to address the prediction problem. P-CAC introduces component performance specification using the concept of atomic operations that are derived using traditional algorithmic analysis. We refine the connector set proposed by Mehta et al. [10] by incorporating the connectors’ properties as part of the interface specifications for reasoning about its performance. A web of interacting processes which is the consequence of the properties and configurations of the components/connectors, is then used to deduce the performance of the composed system. We discuss these findings and suggest further research directions.

1 Introduction

Functional aspects of software components have been the center of attention in the research community for quite some time. Recently, more efforts have been placed in researching the quality aspects of component-based software. This is demonstrated by the organization of component-based software engineering (CBSE) workshops targeting at quality aspects at several international conferences. The recent 6th ICSE workshop on “Automated Reasoning and Prediction” in Portland USA is one of them.

Software performance engineering (SPE) is an important field in real-time and embedded systems. Over the years, efforts have been directed in areas including measurement, simulation, and modeling techniques. Recent years saw the emergence of component software technology in the industry and its fusion with SPE.

In a typical component-oriented engineering situation, the software engineer often faced with the decision of component selection and assembly. Commercial component assembling technologies such as .NET, CORBA, EJB have been well received in the industry. However, none offers the capacity to predict performance. It is desirable to predict the system performance based on various architectural designs.

In this paper, we illustrate our ideas with a web-based system as shown in Figure 1. Each oval represents a component while the dotted and solid arrows are the communication paths. Each connector stream, procedure call, and event instance may have one or more paths. In this example, the event connector comprises of four paths while the rest has exactly one path each.

In this system, a client requests web pages through the broker who then contacts an available server. The server then retrieves data from the database, formats the web page with the data, and returns the results. The broker uses event messages to coordinate the clients and servers; servers talk to the database by sending stream requests; server-to-server interactions involve procedure calls.

Although the assembled system may satisfy its functional requirements, i.e. deliver the correct web page to the client, i.e. one does not know the time elapsed during delivery until the system is constructed and measured. Consequently, the system may fail to cope with the performance demands as a result of an increase in client’s demands and/or introduction of more clients. This then leads to the situation whereby a “fix-it-later” approach ([14]) is
required. If some insights into the system’s performance could be gained before construction, this situation could have been avoided.

![Figure 1. The Web System Built Using Components and Connectors](image)

A framework named Performance Characterisation And Composition (P-CAC) provides this capability. In P-CAC, a rich set of parameterisal software connectors and components are used for system modeling and reasoning. Our preliminary efforts looked at specification of individual component’s performance, connector characterisation, and issues concerning composition using connectors. The aim is to predict the execution time of a system’s task.

The paper is organized as follows: Section 2 presents the general issues addressed by P-CAC. Section 3 describes the Component Performance Specification Language (CPSL) which is used to capture the duration taken by the component’s service. Section 4 outlines the component interface and its properties. Section 5 introduces the stream, procedure call, and event connector types and their characteristics. Section 6 illustrates how concepts in previous sections can be used in a composition setting. Section 7 examines some future research directions while Section 8 discusses related work.

## 2 The P-CAC Framework

The Performance Characterisation And Composition (P-CAC) framework is designed to address the increasing need for an abstract form of compositional performance reasoning about systems consisting of components and connectors. P-CAC’s primary aim is to develop analysis techniques and software tools to assist software engineers in predicting the assembled system’s eventual performance in a deployed environment. P-CAC’s secondary aims are to provide 1) a performance description notation and 2) a set of connectors to assemble system for software performance engineers to use.

The primary entities in P-CAC are components and connectors. Every component has one or many published services. Services may be connected to one another via connectors. Channels are the communication paths established as a result of the specified connection between service pairs. Depending on the connector type used, the information moved along the channels are manifested as streams, messages, or procedure calls. Port provides a logical separation between component’s “bounds” and its environment. The information arriving at or departing from the component will always traverse pass the component’s ports.

Connectors in P-CAC is based on the pattern-based programming paradigm as advocated by Gamma et al. [5] and Buschmann et al. [3]. P-CAC’s connectors offer high-level design patterns that include timing information. The system performance is constrained by the specific properties each connector type possesses.

The P-CAC framework involves the following software lifecycle phases.

- **Software Engineering Phase**: This involves the repetitive cycle of software engineering activities including requirements analysis, design, programming, etc.

- **Performance Characterisation Phase**: In this phase, the engineer derives the abstract time cost of individual services of the developed components and configurations of connectors based on algorithmic analysis. CPSL is then used to express these costs in atomic operations (Section 3). The properties of the required connectors and components are then determined and published in their respective interfaces (Sections 4 and 5).

- **Assembly & Performance Prediction Phase**: System tasks with critical performance requirements are now identified and their performance is predicted by analysing the chosen connectors’ configurations (composition scenarios). This is discussed in Section 6. Since the deployment environment is known, the costs of the atomic operations (of components’ services) can be discovered using a software probe (Section 3). These costs, in addition to the costs that arise from composition, is the total time taken by the system to complete a task.

## 3 The Component Performance Specification Language

The execution time of a component’s service is expressed in the Component Performance Specification Language (CPSL). It allows one to express service’s duration in an approximated number of atomic operations based on program constructs. Each atomic operation is distinct and has an associated cost in certain environment. Their costs are discovered after running the probe that simulates the atomic operations executing in a specific environment. The probe
3.1 Atomic Operations

We strive to achieve the following:

1. have a small set of operations;
2. reasonable tradeoff between accuracy in performance against specification complexity;
3. hide component’s implementation as much as possible;
4. support for platform independence and abstraction away from hardware.

Table 1 lists the atomic operation types needed to express the duration of a component’s service. This set covers the major Java language imperative instructions. Java instructions such as the arithmetic operators are classified together for two reasons: (i) to reduce complexity of duration’s specification and (ii) certain Java instructions have relatively close costs. For example, the atomic operation type named “arithmetic” (abbreviated as \texttt{art} in specification) may refer to any one of the Java instructions: +, -, \times, 1. In CPSL, the Java instructions classified under the same atomic operation type have similar costs. For instance, the addition of two long variables has an average cost of 4.6244 \times 10^{-4} \text{ms} whereas addition of two int gives approximately 1.0693 \times 10^{-4} \text{ms}. These averaged measurements were taken by our probe which runs in Windows 2000 on a Pentium II processor. The probe is written in Java using the Sun’s Java compiler version 1.4.0. If we were to distinguish between long and int additions, it would improve accuracy by approximately 3.5551 \times 10^{-4} \text{ms}. Instead, we chose to sacrifice this precision by trading for specification simplicity.

One should consider the average running time taken by the instructions classified under the same atomic operation type. In this way, the probe is able to provide the values of various atomic operation types in the deployed environment. Take for instance, we simulate the arithmetic instructions which involve a combination of primitive data types such as int, float, long, etc. More precisely, the probe executes the following code:

\begin{verbatim}
int x = 50;
float y = 100;
// start clock
// instructions to simulate arithmetic
// involving int and float
x + y ........ x - y ........
x / y ........ x * y ........
//end clock and record time elapsed
\end{verbatim}

This approach assumes that different occurrence frequencies involving dissimilar usages of arithmetic instructions in the service’s implementation would provide an averaged value for \texttt{art} in the deployed environment.

We now discuss each atomic operation. For array creation, we have \texttt{arr(size)} to represent Java code such as \texttt{x = newint[bufsz]} and \texttt{y = newNewObject[c]}. Parameter size of atomic operation type \texttt{arr} would then refer to \texttt{bufsz} and \texttt{c} respectively. For instance, if \texttt{bufsz} is 10, we then have \texttt{arr(10)}. The \texttt{asm} operation represents the Java byte code instruction Xstore\textsuperscript{1}. Xstore refers to assignment instructions including integer and float assignments (represented by \texttt{istore} and \texttt{fstore} respectively).

The atomic operation type \texttt{ref} represents Java byte code instruction Xload. Variable reference for reading purposes in the code would be translated into Xload by the Java compiler. For instance, ‘g++’ would be translated into the following byte codes: iload (referencing operation), dconst (constant 1, another referencing operation), and dadd (adds up ‘g’ and ‘1’). Thus, we specify two \texttt{ref}s and one \texttt{art} for a statement like ‘g++’.

The comparison (cmp) operation records the cost associated with statements like 'longVariable >= 1000', 'boolVariable == true', etc. We classify the Java logical instructions '>=, ==' as cmp.

Loops and conditionals are also covered by the atomic operations. E.g. while(j < 10) becomes ref (the j), cmp (the <), ref (the constant 10).

<table>
<thead>
<tr>
<th>Atomic Op./Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create new object/cno</td>
<td>The time to create an object, ie. instance.</td>
</tr>
<tr>
<td>Arrays/arr(size)</td>
<td>The time to create an array of members containing primitive types or object references. Parameter size refers to the number of array members.</td>
</tr>
<tr>
<td>String concatenation/cat(stSz1, stSz2,...)</td>
<td>The cost to concatenate strings of various sizes stSz1, stSz2, etc. The parameters are numeric.</td>
</tr>
<tr>
<td>Assignment/asm</td>
<td>The cost to assign an object reference or primitive data type.</td>
</tr>
<tr>
<td>Referencing/ref</td>
<td>The cost to reference an object reference or primitive data type.</td>
</tr>
<tr>
<td>Comparison/cmp</td>
<td>The cost to perform logical instructions, ie. ==, &gt;, etc.</td>
</tr>
<tr>
<td>Arithmetic/art</td>
<td>The Java code that performs arithmetic.</td>
</tr>
</tbody>
</table>

Table 1. Atomic Operation Types

3.2 Specification Example

The following Java method accepts a string and returns true if it contains only alphabets and/or numbers and returns false otherwise. The atomic operations needed are listed together with the code.

```java
public boolean // asm for "String str"
isAlphanumeric(String str) {
    if (str.length() == 0) // ref*2, cmp
        return false;
    // asm; ref, cmp, ref; ref, art
    for (int gg=0; gg < str.length(); gg++) {
        // asm, ref for "String str"
        char aChar = str.charAt(gg);
        if (!Character.isLetterOrDigit(aChar))
            return false;
    }
    return true;
}
```

Parameter len is the length of the string to be processed by this service. The keyword other_duration means its parameter has an associated cost and is defined separately. Thus, like isAlphanumeric, we assume the standard Java methods isLetterOrDigit, length, charAt are defined separately in the later part of this service specification.

```java
service isAlphanumeric : {
    duration(len) := { //fixed cost
        asm*2 + ref*2 + cmp +
        other_duration(String.length())
        + //variable cost
        len * (asm + ref*4 + cmp + art +
            other_duration(loopCost) )
    };
    loopCost := other_duration(String.length())
    + other_duration(String.charAt(int))
    + other_duration(Character.isLetterOrDigit(char))
}
```

3.3 Comments

The idea of using a symbolic cost model is based on the work by Wang [16]. He considers structuring the program symbolically to produce a cost model for compilers to optimize their performance. A set of atomic operations (processor-dependent) are proposed which can be mapped to the intended processor architecture. For our purpose, we adapt this technique to describe portable duration specification.

We briefly discuss why various SPE techniques including measurement and simulation are not suitable for our purpose. Measurement is platform dependent and parameterization is difficult. Furthermore, measurement is a labourious effort; one has to construct and wait for the completion of the services before results are obtained. Similar reasons apply to simulation. Constructing simulation tests are tedious and restrictive in the sense that system emulation are often confined to a limited number of usage and deployment scenarios.

We intend to investigate the accuracy of the published duration by comparing it against the measured data obtained in a deployed environment.

4 Performance Interface of Component

This section discusses the details of component interface. The service’s duration which is written in CPSL, is part of this interface. The reasoning (Section 6) process uses this interface to work out the total execution time of the service as a result of composition.

The structural aspects of a component are services and ports while its properties describe the behavioural aspect. Each component has one or more services, each of which is connected logically to its environment using connectors.
The component has one or more ports which relates data transmission between its services and its external environment. The behavioural description of the ports provides the information concerning how data passed between the environment and component’s services influence the system performance.

We shall use component ClientAlpha of the web system in Figure 1 to explain these aspects. ClientAlpha itself is defined in Figure 3. This component has two ports and two properties named MessageBound and MessageBuffer. It offers service requestRemoteService that requests web pages from a remote server and can be composed using an event connector type that transmits event messages to and from its ports. We now proceed to explain its interface in detail.

Service is an externally visible piece of work offered by the component to its environment. Process-algebra description of service is used to describe how a service operates using process names in sequential/parallel fashions and illustrates how composition is accomplished. We define process as a software operation that performs computation and requires time to execute.

As described in Figure 3, service requestRemoteService prepares information for transmission (process prepare_info), transmits an event message through the event connector (process external_request(out event)), then waits for the reply (process receive(in event)), and finally cleans up resources (process finalize). The binary operator -> when used as p1->p2, indicates process p1 runs to completion before p2 starts. Each process has a cost as indicated by the keyword duration. Processes defined as “p1(in|out information_type)” have additional cost during composition time and is also used to indicate ‘location’ of the composition points. For example, process external_request(out event) accepts an event connector type that transmits outgoing event messages. Besides its own cost, it needs to take into account of the costs incurred by other services when requestRemoteService is composed with them (Section 6).

Ports define the outlet in which information flows into and out of the component. Port outPort1 of component ClientAlpha will discard an event message (the instance evt) once the message size exceeds its buffer. A delay of 30 time units is experienced before the message is discarded. The purpose of accounting for such stimuli is because they will be used in the reasoning process.

Property defines a published characteristic of the component that may be part of the cost of one or more services. We explain how a property plays its role with regard to these costs. The properties of component ClientAlpha in Figure 3 are MessageBuffer and MessageBound. The fixed and variable costs expressing service requestRemoteService may take into account the constraints imposed by these properties through the service’s parameters such as msg_size. Parameter msg_size refers to the size of a single event message to be transmitted by the component in bytes and is numeric. In ClientAlpha, the variable cost of duration for the process external_request describing service requestRemoteService involves msg_size. The purpose of expressing properties as part of interface is to account for their characteristics during reasoning.

```plaintext
component ClientAlpha {  
  outgoing port outPort1;  
  incoming port inPort1;  
  //Description of port’s behaviour  
  outPort1 {  
    buffer size = 200 bytes;  
    on message_arrival_event(evt) do {  
      //discard if buffer full  
      if sizeOf(evt) + current_pool > size  
        delay 30; do_discard;  
      fi  
    }  
  }  
  outPort2 { .... }  
  property MessageBound = 128  
  //Stores if server has not finish  
  //processing request.  
  property MessageBuffer = 1024  

  service requestRemoteService : {  
    //Description of service’s behaviour  
    proc1 := prepare_info ->  
      external_request(out event) ->  
      receive(in event) -> finalize  
    prepare_info : duration :=  
      { ....... }  
    external_request :  
      duration(msg_size) := {  
        //fixed cost  
        arr(10) + 5*cmp + 10*art  
        + //variable cost  
        (6*asm + 12*ref + 4*art)/msg_size  
      }  
    receive() : duration :=  
      { ....... ....... }  
    finalize : duration := {  
      30*ref + 14*art  
    }  
  }  
}
```

Figure 3. Definition of Component ClientAlpha

5 Connector Types

P-CAC uses the primitive connector types listed in the connector taxonomy proposed by Mehta, Medvidovic and Phadke in [10]. This taxonomy is intended to serve as a framework to promote the awareness of connector status. It includes Procedure Call, Event, Stream, Data Access, Linkage, Arbitrator, Adaptor, and Distributor connector types.
Due to space limitation, we confine ourselves to the first three types for discussion. Each connector type has several dimensions. stream connector has bounds, buffering, delivery, format as its dimensions. In P-CAC, we refer these dimensions as properties.

In this section, we discuss how Procedure Call, Stream, and Event can be characterized based on their properties.

5.1 Event, Procedure Call, and Stream Connectors

The structural aspects of a connector are properties and configurations. A configuration is a particular topology that specifies how services are connected by the connector. Essentially, it serves as an interaction protocol between components and it involves delays introduced by the connector’s implementation. The connector’s properties may influence the performance of its configurations. Each connector type is characterized by its own specific properties.

Process-like operators are used to describe how services of components are composed together. One such operator ->, has already been introduced briefly in Section 4. The set of operators are

- p1->p2 indicates p1 runs to completion before p2 starts;
- p1 /- p2 states that p1 invokes p2 asynchronously;
- p1 -/ p2 indicates p1 invokes p2 synchronously, where p1 and p2 are processes.

Description of time delay or process with/without connector properties, may follow after each operator. Eg., “p1 -|40 p2” means there is a delay of 40 time units in between p1 and p2 while “p1 -| delay(x,y,z) p2” indicates transmission between p1 and p2 are characterized by the connector properties x, y, and z. “p1 -/ delay p2” means connector properties are not involved during the execution of p1 and p2.

We demonstrate how individual connector types can be expressed in P-CAC, beginning with the event connector, follow by the procedure call and the stream. The connector definition for ClientBrokerServerConn of event type is listed in Figure 4. The property of our event connector type is the notification scheme. This scheme indicates which component will play the coordination role to realize event forwarding and replication such that functional requirements are satisfied.

ClientBrokerServerConn has a configuration main which provides a selective notification scheme with component Broker forwarding a client request to a single server (one_to_one). A selective scheme means that the component Broker will depend on some constraints during server selection. This is indicated by the clause “notification_scheme = Broker + selective + one_to_one”.

The process client_request_page in configuration 1 indicates which processes of different services are collaborating to realize a single request for web page by ClientAlpha. The details of client_request_page consist of a series of constructs like “component.service.process "X"”, each denoting a specific process sending out message X. Operator “-“ means transmission. Each “component.service.process "X"” is composed with another using the operators introduced earlier. Thus, “(ca.requestRemoteService.external_request “msg1) -|...“ reads as external_request of ClientAlpha’s requestRemoteService transmits message named msg1 to wait of Broker’s forwardToServer synchronously. When wait of Broker’s service forwardToServer runs to completion, forward then transmits msg2 to ServerMercury. The operators () group the processes within the same component’s service together while each member of the set ->, -| and -/ associates left to right. The rest of client_request_page is interpreted similarly.

The delays experienced during forwarding of events between different components are reflected by the ClientBrokerServerConn’s processes delay_ca_bk, delay_bk_sm, delay_sm_bk, and delay_bk_ca. These processes are the connector’s cumulative “localized” costs.

connector ClientBrokerServerConn event_type {
  let ClientAlpha, Broker, ServerMercury as ca, bk, mm;
  configuration main {
    notification_scheme = Broker + selective + one_to_one;

    client_request_page :=
      (ca.requestRemoteService.external_request "msg1) -| delay_ca_bk
      (bk.forwardToServer.wait -> bk.forwardToServer.forward "msg2)
      -/ delay_bk_sm
      (sm.getWebResponse.wait -> sm.getWebResponse.forward "msg3)
      -/ delay_sm_bk
      (bk.forwardToClient.forward "msg4)
      -/ delay_bk_ca
      (ca.requestRemoteService.receive);

    delay_ca_bk : duration()
      := { cno*12 + cmp*3 + art*32 }
    delay_bk_sm : duration()
      := { /*in atomic operations*/ }
    delay_sm_bk : duration()
      := [ .... ]
    delay_bk_ca : duration()
      := [ .... ]
  }
}

Figure 4. Definition of an Event Connector
Figure 5 shows a procedure call connector named \texttt{ServerToServer} that offers two configurations, the \textit{procedure call} and \textit{callback}. Implementation pattern is a property of our procedure call connector type. In this example, each implementation pattern, i.e. callback, is manifested as a single configuration.

The first configuration has \texttt{sendUpdate} invoking \texttt{receiveUpdate} synchronously and this call takes 5 time units (call overhead time) before the latter starts execution. The second configuration indicates an asynchronous invocation (callback registration by \texttt{ServerMercury's processReq}) followed by callback by \texttt{ServerSaturn's returnData}. The registration has to run to completion before \texttt{returnData} makes a callback.

```plaintext
connector ServerToServer proc_call_type {
    let ServerMercury, ServerSaturn as sm, ss;
    configuration call {
        sm.updatePeer.sendUpdate -|5
        ss.updateService.receiveUpdate;
    }
    configuration callback {
        //Register
        sm.getWebPage.processReq ->
        ss.retrievalService.requestData;
        //Callback
        ss.retrievalService.returnData ->
        sm.getWebPage.format;
    }
}
```

Figure 5. Definition of a Procedure Call Connector

The stream connector shown in Figure 6 transmits data blocks between \texttt{returnData} and \texttt{retrieveData} and its properties are \textit{stream block size} and \textit{transmission frequency}. \textit{Stream block size} refers to the size of a logical data block in bytes being transmitted along the stream whereas \textit{transmission frequency} states the number of these blocks transmitted per unit time. These properties influence the connector’s performance by varying the number of atomic operations needed for this connector to complete its own internal computation. Process \texttt{delay} describes this internal computation.

```plaintext
connector ServerToDB stream_type {
    let ServerSaturn, DataBase as ss, db;
    configuration main {
        ss.retrievalService.returnData
        -| delay(block_size,trans_freq)
        db.accessData.retrieveData;
        delay : duration(block_size,trans_freq)
        := { arr*40 + ref +
            cmp*block_size + art* trans_freq }
    }
}
```

Figure 6. Definition of a Stream Connector

like previous efforts by others, our work incorporates various connector types and their composition alternatives, which facilitates time cost reasoning at a richer and abstract level.

Classical process algebras such as Communicating Sequential Processes (CSP) and \(\pi\)-calculus could have been used in place of the operators introduced in Section 5.1. However, they do not model the time concept. Although timed CSP has been proposed, we have not investigate whether it is suitable to express the interaction semantics. We find it necessary to introduce explicit process description of services for composition purposes because the execution of a service usually demands external services of other components. If we do not expose some of the service’s workings, we are restricted to the fact that a service can access external services only 1) when it has run to completion or 2) before it has been invoked, either which is impractical in real applications. We have explored a small set of properties including stream bounds and implementation patterns. Other interesting properties such as delivery of events and streams (best effort, at-least-once, etc.) will be considered in the future.

6 Composition Performance Reasoning

During the “Performance Characterisation Phase” (Figure 2), we specify the performance of components using the material presented in Sections 3 and 4. Connectors are engineered and specified as discussed in Section 5. During the “Assembly & Performance Prediction Phase”, we perform reasoning based on a pool of components and connectors. No actual assembling is required in P-CAC until the system’s performance is deemed satisfactory.

In this section, we highlight the issues for consideration concerning the derivation of a task’s execution time. For illustration, we assume the client’s web page request is a single task. In other words, determine the amount of time taken by component \texttt{ClientAlpha's requestRemoteService}. 
The steps performed during the “Assembly & Performance Prediction Phase” are

1. Select a set of components and connector types.

2. Composition: Set the properties of the connectors. Eg., supply values for parameters block_size, trans_freq in Figure 6.

3. Iterate the following steps until satisfactory performance is obtained:
   
   (a) Reason about the performance of the task. In our case, requestRemoteService.
   
   (b) If performance is satisfactory, quit. Otherwise, perform one or more of the following activities.
      
      • Re-engineer components/connectors.
      • Add/remove components/connectors.
      
      Repeat step 3a.

In step 1, the selected components are ClientAlpha, ClientBeta, ServerMercury, ServerSaturn, and DataBase (defined in Figures 3 and 8). Partial component interface descriptions are given in Figure 8 to complement our discussion. The connector types used are an event connector (Figure 4), procedure call connector with configuration callback (Figure 5), and stream connector (Figure 6).

In step 2, the engineer decides on the values for parameterisable properties of the stream connector in Figure 6 and msg_size of ClientAlpha component in Figure 3 for investigation.

Using the composition details given in Figures 4, 5, and 6, we can now derive the interaction diagram shown in Figure 7 intuitively. This diagram shows the entire process flow for requestRemoteService when ClientAlpha is composed with the rest via connectors. The interaction diagram follows the structure of UML sequence diagram. However, unlike sequence diagram, our interaction diagram depicts 1) information passed between components and 2) services and processes belonging to various connectors. Each rectangle, black dot, dotted line represents component, process, and channel respectively. The vertical timeline is interpreted similarly as in UML sequence diagram. Each channel is annotated with information type, if any, followed by process operators -| and -/. The vertical brackets denotes the encompassed processes belonging to individual services while horizontal ones show the connected components by connectors.

In step 3a, we work out the total time taken by requestRemoteService. Since each process in the interaction diagram has an associated cost given in their respective interfaces, the overall time can then be predicted. Although we have not investigate how this cost can be systematically derived, nonetheless, we have identified some related issues. Firstly, the synchronicity between processes (−|, −/, etc.) has to be resolved. Second, we need to include the port’s delay at various components such as outPort1 and inPort1 of ClientAlpha (Figure 3). Third, the information type moving between components such as msg1 between ClientAlpha and Broker has to be taken into consideration. For example, message with different priorities arriving at Broker.forwardToServer.wait may require different execution times. Although we did not capture this in the interface explicitly, the possibility of modeling various information types as a factor for reasoning is retained for further investigation.

The remaining step 3b involves experiment with various composition alternatives, configurations, and properties to achieve performance objective.

6.1 Comments

Our preliminary effort on performance reasoning is based on the processes defined in their respective interfaces before being employed in a system-wide composition. We plan to explore these ideas further and investigate a solution to accommodate incremental reasoning based on them. The introduction of new components/connectors should not
component Broker {
  service forwardToServer :
    { procl := wait(in event) ->
      forward(out event) -> procl }
  service forwardToClient :
    { procl := wait(in event) ->
      forward(out event) -> procl }
}

component ServerMercury {
  service getWebPage :
    { proc1 := wait(in event)
      -> processReq(out proc_call)
      -> format(in proc_call)
      -> forward(out event) }
}

component ServerSaturn {
  service retrievalService :
    { proc1 := wait(in proc_call)
      -> requestData(out stream)
      -> returnData(out proc_call) }
}

component Database {
  service accessData :
    { proc1 := retrieveData(in stream) }
}

Figure 8. Partial Definition for Components Broker, ServerMercury, ServerSaturn, and DataBase

warrant for reexamination of those already analyzed.

Traditional modeling formalisms could be used instead of the interaction diagram. Real-time UML [11] offers guidelines on modeling time and performance concepts in UML but nothing on compositional deduction of interacting processes. Stochastic process algebra (SPA) provides parallel/sequential timed actions as the modeling concepts. Although the processes in Figure 7 can be modeled using SPA, SPA's capability to model properties such as stream's block size (Figure 6) needs further investigation.

We intend to explore these ideas and options further.

7 Potential Research Directions

This section points out some limitations in our work done so far and the areas for future research.

Our component’s interface does not account for stochastic and probabilistic delays. Also, the component/connector characterisation did not take into account of external environmental factors and resource constraints, including operating system policies (caching policies etc.), execution constraints imposed by amount of memory and disk latency, and synchronisation overheads.

Characterisation of components becomes complex when environmental factors are considered. Firstly, resource requirements have to be explicitly introduced as part of the interfaces, ie. how much a service require of memory, disk, etc. Secondly, factors such as operating system policies have to be clearly defined before they can be incorporated into P-CAC for modeling and reasoning. For the same reasons, the relationships between the properties (buffer, stream bounds etc.) and resources have to be investigated too.

An empirical-based approach by Chen et al. [4] takes some of these factors into account by profiling and benchmarking applications. Although the approach is relevant for true black-box components, its measurement-based approach is not suitable for highly parameterisable components (Section 3.3).

8 Related Work

The idea of using atomic operations in the proposed CPSL is based on other symbolic cost models including the PAMELA [15], Wang’s approach in [16], and the approach by Liu and Gomez [9]. Liu, Gomez, and the PAMELA group study the effects of inputs to program functions and path analysis based on the sequential/parallel composition of symbolic constructs. However, none is concern about the expression of portable component performance interface in a deployed environment.

Sitaraman et al. [13] investigated formal reasoning about component’s performance specifications concerning their space requirement and time costs. Based on code analysis, the workings of the component are augmented with the data types explicitly. Unlike their approach which exposes the entire component’s workings, our approach involves only partial exposure through the abstract atomic operations. Our design also accommodate composition and properties, both of which they did not consider.

Balsamo, Inverardi, and Mangano [1] presented an approach to transform software architecture description in CHAM (Chemical Abstract Machine) to an open queueing network model for performance evaluation. The strength of this approach is the flexibility of analyzing various automatically derived QN from various competing software architecture specifications. However, some restrictions are noted. The notion of separation between connectors and components is unclear in CHAM. As a result, introduction of new component or connector is difficult because relevant interface specification does not exist. Parameterizable, contractual specification is not supported at the CHAM level. Instead, parameterisation is decided at the QN level.

The framework introduced by Hissam et al. [6] PECT (Prediction-Enabled Component Technology) attempts to achieve prediction about functional and non-functional objectives which include latency. Through iterative refinement of component assemblies, performance measurements are then estimated. PECT proposes a solution to compute latency based on dependencies between composed compo-
nents. The PECT approach is similar to P-CAC: acquire components, virtual assembling, analysis of non-functional properties. However, their analysis does not involve software connectors nor incremental composition of components.

The PACC (Predictable Assembly from Certifiable Components) by Larsson et al. [8] is a framework that supports predicting certain properties such as versioning and response time estimation. Derivation of end-to-end response time is based on component relations and interaction patterns such as mutual exclusion. Although PACC does consider interaction patterns, our approach goes a step further by examining a broad variety of connection types.

The approach used by Jayaputera, Poernomo, and Schmidt in [7] is targeted at reasoning about the reliability of the composed system. By having the system description transformed into relevant PFSMs (probabilistic finite state machines), based on the selected fault-tolerant algorithm, a final PFSM is constructed from sub-PFSMs in a compositional fashion. PCTL (Probabilistic real-time computational tree logic) technique which facilitates model checking is used to deduce system reliability. Their approach is similar to P-CAC in terms of the final objective: reasoning properties at the system level. A major difference from our work is automata are used instead of processes for reasoning.

9 Conclusion

Research in system-level performance prediction of assembled COTS is still a relatively new area. To the best of our knowledge, no one has yet considered a ‘rich’ set of connectors in a composition setting.

We have presented and discussed our performance prediction framework P-CAC and its related issues. The next immediate step is to work out a solution to reason about the performance of the composed system as described in Section 6. The prediction results will be compared with an actual web-based system to determine the prediction accuracy.

References

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.