Prediction of run-time consumption in multi-task component-based software systems

Prediction of Run-time Resource Consumption in Multi-task Component-Based Software Systems

Johan Muskens and Michel Chaudron

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
J.Muskens@tue.nl, M.R.V.Chaudron@tue.nl

Abstract. Embedded systems must be cost-effective. This imposes strict requirements on the resource consumption of their applications. It is therefore desirable to be able to determine the resource consumption of applications as early as possible in their development. Only then, a designer is able to guarantee that an application will fit on a target device.
In this paper we will present a method for predicting run-time resource consumption in multi-task component-based systems based on a design of an application. In [5] we describe a scenario based resource prediction technique and show that it can be applied to non-pre-emptive non-processing resources, like memory. In this paper we extend this technique, which enables us to handle pre-emptive processing resources and their scheduling policies. Examples of this class of resources are CPU and network.
For component based software engineering the challenge is to express resource consumption characteristics per component, and to combine them to do predictions over compositions of components. To this end, we propose a model and tools, for combining individual resource estimations of components. These composed resource estimations are then used in scenarios (which model run-time behavior) to predict resource consumption.

1 Introduction

Our research was carried out in the context of the Robocop project[1]. The aim of Robocop is to define an open, component-based framework for the middleware layer in high-volume embedded appliances. The framework enables robust and reliable operation, upgrading, extension, and component trading. The appliances targeted by Robocop are consumer devices such as mobile phones, set-top boxes, dvd-players, and network gateways. These devices are resource (CPU, memory, bandwidth, power, etc.) constrained and the applications they support typically have real-time requirements. The component model used within Robocop will be introduced in section 2.

Resources in embedded systems are relatively expensive and (usually) cannot be extended during the lifetime of the system. Consequently, an economical pressure exists to limit the amount of resources, and applications have strong constraints on resource consumption. In contrast to the fixed nature of available resources, software is subject to change. For example, new applications / components can be added, or existing ones can be replaced or removed.

To make sure that a combination of applications fits on a particular target system, knowledge about their resource consumption is required. Access to this information at design-time can help to prevent resource conflicts at run-time. This helps to decrease development time and, consequently, leads to a reduction of development costs.

However, the current techniques for addressing non-functional aspects of a system during the early phases of software development are laborious. In this paper we describe a method for estimating resource properties for component-based systems, using models of the behavior and the resource consumption of individual components. Key features of our approach are that the prediction method can be used in the early phases of application development, is intuitive, and requires little effort.

[1] The project is funded in part by the European ITEA program. This is a joint project of various European companies, together with private and public research institutes.

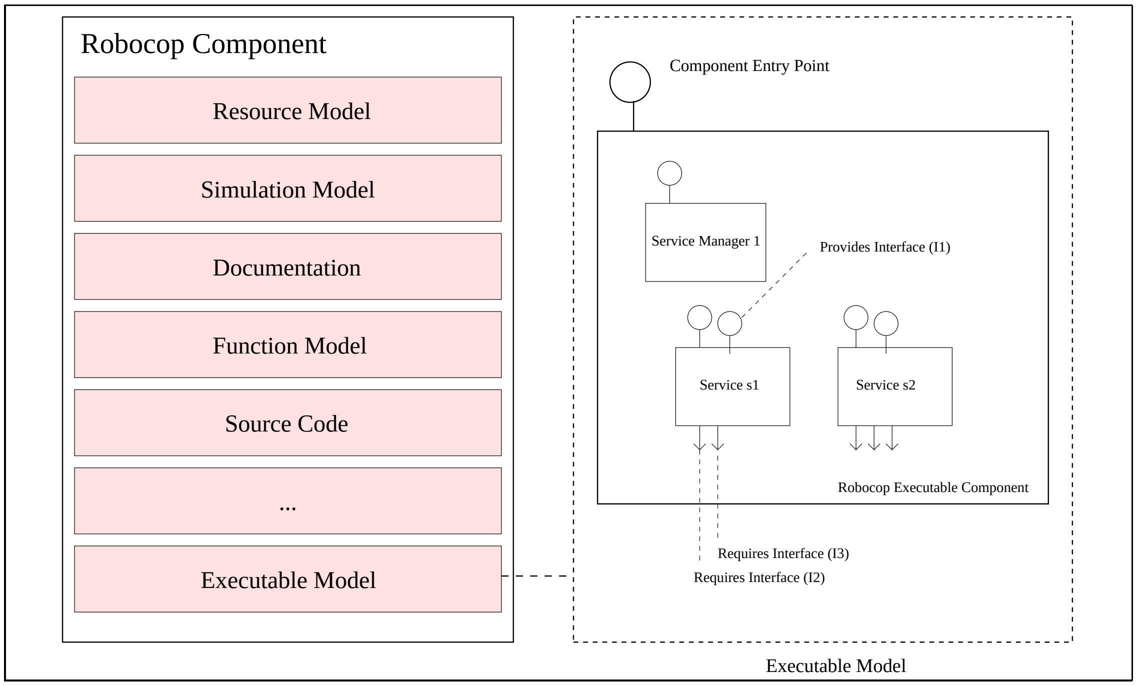

Fig. 1. Robocop component model.

We consider applications that may consist of multiple tasks. Tasks define a logical flow of control. Tasks may run across multiple components. We associate a behavior model with each component. A behavior model of a task is obtained by composing the behavior models of individual components. We associate a resource model with operations of components. Using these behavior models for the tasks, the resource model for operations and a scheduling function we are able to predict the resource consumption for the system.

We propose a scenario-based prediction method in order to avoid the combinatorial complexity of full state-space analysis of systems, as is encountered by model-checking approaches [2]. In our approach resource estimations are made for a set of scenario's that represent plausible/critical usages/executions of the system. Similar to the use of scenario's in the evaluation of software architectures [12], the validity of the results of our analysis method depends on the selection of the scenario's that are used.

The paper is structured as follows. In Section 2 we discuss the Robocop component model. In Section 3 we make a classification of different resource types. In Section 4, we present the theory for scenario based resource prediction. In Section 5 we show how to apply our method in practice, by showing an example. Section 6 discusses related work and draws some conclusions.

2 The Robocop Component Model

The Robocop component model is inspired by COM [1], CORBA [13] and Koala [14]. A Robocop component is a set of models. (see figure 1). Each model provides a particular type of information about the component. Models may be in human-readable form (e.g. as documentation) or in binary form. One of the models is the 'executable model', which contains the executable component. Other examples of models are: the resource model, the functional model, the behavior model, and the simulation model. The Robocop component model is open in the sense that new types of models can be added.

A component offers functionality through a set of 'services'. (see figure 1). Services are static entities that are the Robocop equivalent of public classes in Object-Oriented programming languages. Services are instantiated at run-time. The resulting entity is called a ‘service instance’, which is the Robocop equivalent of an object in OO programming languages.

A Robocop service may define several interfaces. We distinguish ‘provides’ interfaces and ‘requires’ interfaces. The first defines the operations that are implemented by a service. The latter defines the operations that a service needs from other services.

In Robocop, as well as in other component models, service interfaces and service implementations are separated to support ‘plug-compatibility’. This allows different services, implementing the same interfaces, to be replaced. As a consequence, the actual implementations to which a service is bound do not need to be known at the time of designing a service. This implies that resource consumption cannot be completely determined for an operation, until an application defines a specific binding of the requires interfaces of service instances to provides interfaces of a service instance.

Within the Robocop component model tasks are not confined to services. Tasks may, but do not need to, cross service boundaries. An example of a system built up out of multiple services with tasks that run across service boundaries is given in figure 7.

3 Resource Categories

We model different types of resources in different manners. First of all we distinguish processing resources and non-processing resources. Processing resources are resources that process elements; for instance instructions (CPU), packets (Network) and audio samples (Sound card). Other resources are classified as non-processing resources. An example of a non-processing resource is memory. Secondly we distinguish pre-emptive resources and non-pre-emptive resources. Non-pre-emptive resources are claimed by a consumer and can only be used again when they are not needed any more by the specific consumer. The use of a pre-emptable resource may be interrupted before the consumer has finished using the resource. Figure 2 illustrates that pre-emptive resources can be used by multiple consumers at the same time and non-pre-emptive resources can only be used by one consumer at any specific moment in time. Furthermore figure 2 shows that non-processing resources are requested and released by a consumer and processing resources are requested and used until processing is completed. Different types of resources require different type of specifications mechanisms for the usage of the resource. Non pre-emptive resources have a claim mechanism. Pre-emptive resources have a mechanism to indicate the desire to use them (hopefully the resource manager will grant the wish). Non-processing resources have start-using:stop-using mechanism, processing resources only have a start:using mechanism (it is free again when it has processed all the data), there is no stop mechanism. However sometimes there is a notion of a deadline. Table 1 describes how we specify the resource use for different resource types. In this specification we assume resources are countable. The variables X and Y represent a amount of resources.

	Processing	Non-processing
Pre-emptive	Require: X	Require: X, Release: Y
Non-pre-emptive	Claim: X	Claim: X, Release: Y

Table 1. Resource Specification

4 Resource Prediction Method

We want to determine the resource usage of component-based systems during their execution. This run-time resource use depends on the composition of the services that form a system and the operations executed. In this section we describe an approach for predicting the resources that a system uses for a specific scenario built out of multiple tasks. Tool support has been developed for this approach in order to automate prediction of resource consumption.

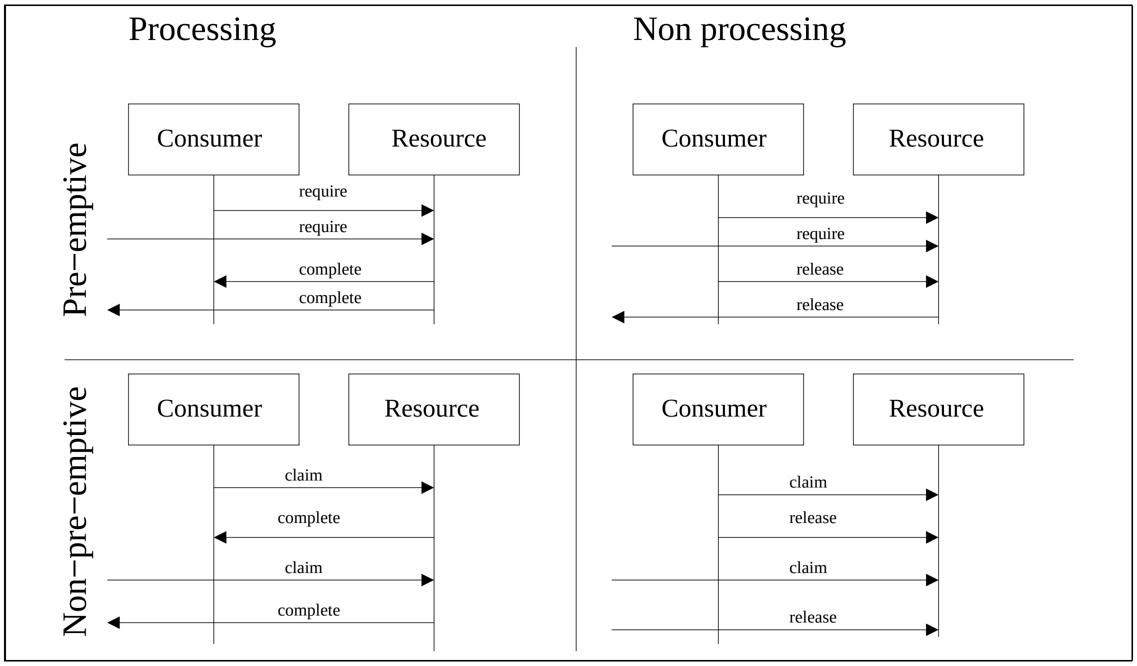

Fig. 2. Resource Categories

This section is structured as follows. In subsection 4.1 we discuss the specification of a service. In subsection 4.2 we show how to determine the set of services needed by an application. Subsection 4.3 shows how we obtain the resource consumption of a specific operation of a service. In subsection 4.4 we discuss how to model resource consumption for call sequences. Finally we consider the resource prediction per task in subsection 4.5 and per scenario in subsection 4.6.

4.1 Service specifications

Our approach considers composition of services. Each service has one or more provides interfaces and zero or more requires interfaces. A provides interface lists the operations that the service offers to other services. A requires interface lists the operations that a service needs in order to function. For the purpose of resource prediction, the resources that are claimed and released are specified per operation.

We propose service specifications to capture such information about services. A service specification contains the following information:

– *provides interface*: A list of interface names;
– *requires interface*: A list of interface names;
– *Resource consumption*: For each implemented operation, the resource usage is specified. The specification of the resource consumption depends on the type of resource (See table 1);
– *Behavior*: For each implemented operation, the sequence of operations it uses.

Figure 3 contains an example service specification. It is a specification for service s_1 which requires the interfaces I_2 and I_3 and provides the interface I_1. Service s_1 implements the operation f which uses operations g and h from interface I_2 and I_3, respectively. Operation f requires 5000 cpu cycles, not counting the invocations of $I_2.g$ and $I_3.h$. Furthermore we can see that operation f claims 100 bytes of memory and on return, releases 100 bytes.

The behavior section of the service specification defines that operation g from interface I_2 is called zero or more times *before* operation h from interface I_3, and that it is called a varying number of times. How such a sequence of calls will be instantiated, in order to predict memory

```
service s1
 requires I2
 requires I3
 provides I1 {
  operation f
   resource cpu
            require 5000
   resource memory
    claim   100
            release 100
   behavior:
    I2.g*;
    I3.h}
```

Fig. 3. An example service specification

consumption, will be discussed shortly. All resource consumption in our approach, is specified per operation. Therefore, resources consumed during service instantiation and resources released at service destruction are specified as part of constructor and destructor operations. The behavior of an operation can be depicted as a message sequence chart (MSC) [11]. Observe that these are partial message sequence charts, because indirect operation calls (for instance operations that are called by $I_3.h$ in Figure 3) are not specified in the service specification.

The service specifications can be defined at design time, or can partly be generated from an implementation. In the latter case, *call graph extraction* can be used to obtain the behavior and to determine the requires interface. In case a service specification was defined at design-time, service specification generation can serve to validate an implementation (i.e., to check whether the service specification, extracted from the implementation, corresponds to the one defined at design-time)

4.2 Composition of services

As illustrated in Figure 3, services can only use operations from interfaces, not directly from services. In other words, in our component model, a service cannot define a dependency upon a specific service. At development- or run-time the decision is made which services to bind to each required interface.

Two services that have the same provides interface may have different requires interfaces. As a consequence, the behavior of an application is not known before all requires interfaces of its services are bound. Hence, this behavior is needed for making estimations of the resource use.

To construct the behavior model of an application, we have to determine and bind the required interfaces. Finding the required services works by transitively replacing required interfaces by services. To describe this process of composition formally, we define a function C. Given a set of required interfaces and service specifications, functions C yields a composition of needed services. The input domain of C is a set of interfaces. We take the output of C a set of service names. In general the output of C may contain more information about the bindings between the services. Function C is defined inductively next. First we introduce some auxiliary notation:

$$\begin{aligned} s.provides &= \text{set of interfaces provided by } s \\ s.requires &= \text{set of interfaces required by } s \\ f.calls &= \text{ordered set of functions called by function } f \end{aligned}$$

$$C(I_1, \ldots, I_n) = C(I_1) \cup \ldots \cup C(I_n) \tag{1}$$

$$C(I) = s \cup C(I')\text{for a service } s \tag{2}$$
$$\text{where } I \in s.provides \text{ and } I' = s.requires$$

If interfaces are provided by multiple services, the set of services might be ambiguous. In this case there are (at least) the following options:

- The composition function C needs assistance to choose between appropriate candidate services. Assistance can be in the form of user intervention (i.e., a person chooses among candidates), in the form of composition constraints, or composition requirements (for instance, a decision is made based on maximal performance, or minimal memory consumption requirements).
- The system generates all possible configurations. This can be used to compare their performance in the next phase.

4.3 Per-operation resource prediction

The set of services established by C, is the basis from which we will be able to deduce the behavior of specific scenarios can be constructed. Using this set we will determine the resource use of called operations. To that end, we define a *weight* function w using 'resource combinators'. This function calculates consumption of a resource r of an operation f over a composition of services, by accumulating the resource usage of f itself, *and* of the resource consumption of all the operations that are called by f. This means that w will compose the behavior triggered by operation f (see figure 4).

The definition of this function depends on the type of resource (see figure 2). Function w is defined as:

$$\begin{aligned} w(f(s),r) = (a) \text{ where}& \\ a = s.f.r.require& \text{ and} \\ r = \textit{pre-emptive and processing}& \end{aligned} \tag{3}$$

$$\begin{aligned} w(f(s),r) = (a,b) \text{ where}& \\ a = s.f.r.require& \text{ and} \\ b = s.f.r.release& \text{ and} \\ r = \textit{pre-emptive and } \neg\textit{processing}& \end{aligned} \tag{4}$$

$$\begin{aligned} w(f(s),r) = (a) \text{ where}& \\ a = s.f.r.claim& \text{ and} \\ r = \neg\textit{pre-emptive and processing}& \end{aligned} \tag{5}$$

$$\begin{aligned} w(f(s),r) = (a,b) \text{ where}& \\ a = s.f.r.claim& \text{ and} \\ b = s.f.r.release& \text{ and} \\ r = \neg\textit{pre-emptive and } \neg\textit{processing}& \end{aligned} \tag{6}$$

$$\begin{aligned} w(f(s_1,\ldots,s_n),r) = {} & w(f(s_i),r) \oslash \\ & (w(f_1(s_1,\ldots,s_n \backslash s_i),r) \\ & \oplus \ldots \\ & \oplus\ w(f_k(s_1,\ldots,s_n \backslash s_i),r) \\ &) \\ & \text{where } s_i \text{ provides } f \text{ and} \\ & (f_1,\ldots,f_k) = s_i.f.calls \end{aligned} \tag{7}$$

The following example illustrates how the functions C and w compose the behavior triggered by

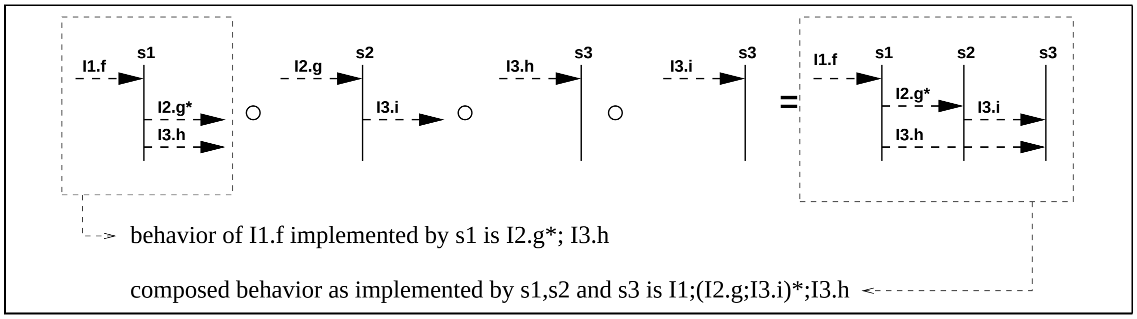
I1.f
s1
I2.g*
I3.h
I2.g
s2
I3.i
I3.h
s3
I3.i
s3
I1.f
s1
s2
s3
I2.g*
I3.i
I3.h
behavior of I1.f implemented by s1 is I2.g*; I3.h
composed behavior as implemented by s1,s2 and s3 is I1;(I2.g;I3.i)*;I3.h

Fig. 4. Composition of behavior.

operation f, following the example depicted in figure 4. The first step is establishing the set of services that form the implementation for the implementation for the operation $I1.f$.

$$w(I1.f(C(I1)), r) = w(I1.f(s_1, s_2, s_3), r)$$

Using this set we are able to compose the behavior of $I1.f$ in 3 steps:

$$\begin{aligned} w(I1.f(s_1, s_2, s_3), r) &= w(I1.f(s_1), r) \oslash \\ &\quad (w(I2.g^*(s_2, s_3), r) \oplus \; w(I3.h(s_2, s_3), r)) \\ w(I2.g^*(s_2, s_3), r) &= w(I2.g^*(s_2), r) \oslash \\ &\quad \big(w(I3.i(s_3), r) \big) \\ w(I3.h(s_2, s_3), r) &= w(I3.h(s_3), r) \end{aligned}$$

Considering resource usage of the execution of two arbitrary operations o_1 and o_2, we distinguish the following cases:

– o_1 and o_2 are executed sequentially: This means that first operation o_1 is executed and after o_1 terminates, then o_2 is executed. An example of two operations that are executed sequentially are $I2.g$ and $I3.h$ in the implementation of interface $I1$ by service $s1$ (see figure 4).
– o_2 is executed as part of the execution of o_1: This means that first operation o_1 is executed. Before o_1 terminates, as part of the behavior of o_1, o_2 is executed. An example of an operation that is executed as part of another operation is $I3.i$, which is executed as part of the the behavior of operation $I2.g$ implemented by service $s2$ (see figure 4).

In these different cases we want to combine the resource usage of the operation executions in different ways. In order to be able to combine the resource usage of operation executions appropriately for each of the cases mentioned before, we introduce a resource combinator for each case. The resource combinator $\oplus$ is used to model resource usage of two sequentially executed operations. The combinator $\oslash$ is used to model resource usage as part of an operation. The definitions of these combinators may be varied, in order to deal with different types of resources or to measure resource usage in different ways. In [5] we considered memory usage and defined these operators, combining claim and release tuples, as follows (see also figure 5):

$$\begin{aligned} (a, b) \oplus (c, d) &= (a - b + c, d) \text{ if } c \geq b \\ &\quad\; (a, b + d - c) \text{ if } c < b \\ (a, b) \oslash (c, d) &= (a + c, b + d) \end{aligned}$$

In section 5 we describe an example concerning cpu usage. In that example we will use the following definitions for the resource combinators, combining cpu claims (see also figure 6):

$$\begin{aligned} a \oplus b &= a + b \\ a \oslash b &= a + b \end{aligned}$$

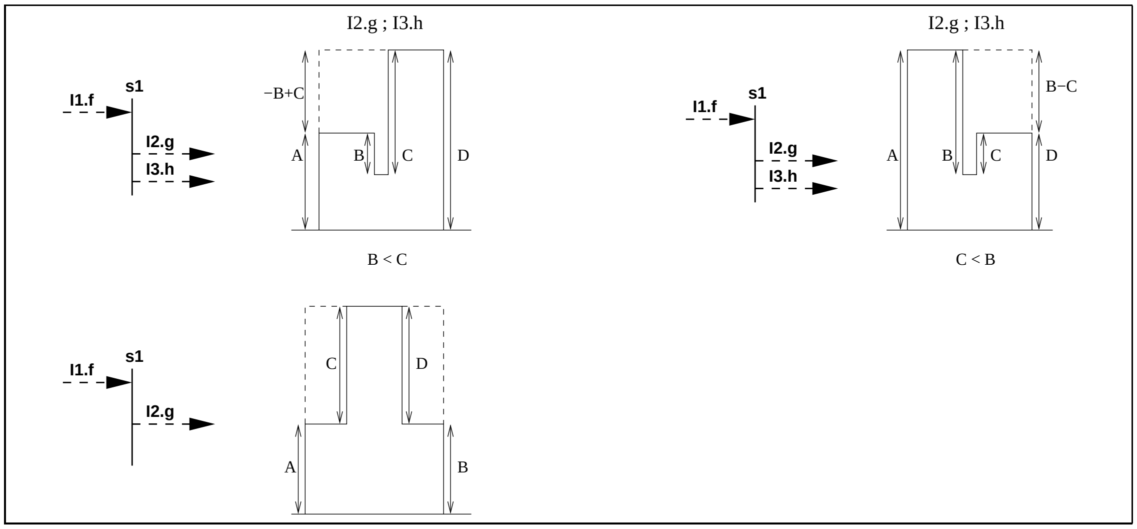

Fig. 5. Example resource combination for memory usage

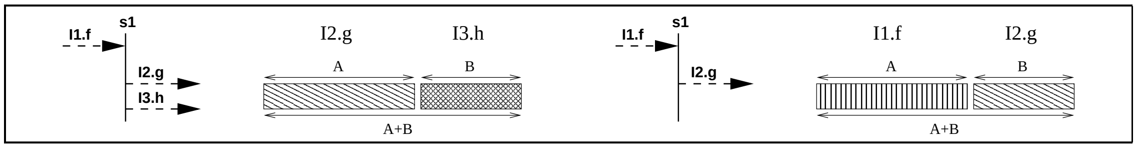

Fig. 6. Example resource combination for cpu usage

Now we combine the composition function C and weight function w, to form the weight function W_{op}, that calculates resource consumption that results from executing an operation of an interface. It is defined as:

$$W_{op}(f(I), r) = w(f(C(I)), r) \qquad \text{for interface } I \text{ and operation } f \tag{8}$$

This functions estimates the resource consumption of resource r of an operation call f from interface I. The function first determines a set of services that (together) form an implementation of f. Then, the weight function w is used to predict resource consumption of f and all operations that are transitively invoked by f.

4.4 Modeling resource consumption for call sequences

Each operation call that is specified in a service behavior specification may be accompanied with an iterator ('$\star$') to indicate that the operation may be called multiple times. For the purpose of resource estimation, we represent an iterated operation call as a lambda expression:

$$f\star = \lambda l \rightarrow l \times f \tag{9}$$

This lambda expression states that operation f is called l times, where l is variable. The weight function for an iterated operator call can now also be defined as a lambda expression:

$$\begin{aligned} w(\lambda l \rightarrow l \times f(s)) &= \lambda l \rightarrow l \times w(f(s)) \\ &= w(f(s))_1 \oplus \ldots \oplus w(f(s))_l \end{aligned} \tag{10}$$

Such lambda expressions are instantiated with concrete numbers, when resource consumption for a complete scenario is predicted.

4.5 Per task resource prediction

As part of the resource prediction for a scenario we need to be able to predict resource consumption for a sequence of operation calls. We introduce the notion of a task, which is a sequence of operation calls. A task can be continuous or terminating, a continuous task corresponds to a repeating sequence of operation calls and a terminating task corresponds to a finite sequence of operation calls.

Using function W_{op} we are able to estimate resource consumption for individual operation calls, we need to predict resource consumption of a task. To that end, we need to know what plausible sequences of operation calls are. Such sequences (which we call task definitions), are determined together with implementors of services. If iterated operations are called in a task definition, the task definition becomes a lambda expression. Defining a task therefore consists of the following two steps:

1. Defining a plausible sequence of operation calls;
2. Instantiating concrete numbers for the lambda expressions, if any.

Instantiating a lambda expression fixes the number of operation calls. Thus, instantiating a lambda expression $\lambda l \to l \times f$ with, say, 2 yields a sequence of two calls to f.

We can now define the weight function W_t for a task, consisting of sequential operation calls. This function computes for each task, e.g. a sequence of operations, the required resources.

$$\begin{aligned} W_t(I.f, r) &= W_{op}(f(I), r) && (11)\\ &\quad \text{for interface } I \text{ and operation } f \\ W_t(g;h, r) &= W_t(g, r) \oplus W_t(h, r) && (12)\\ &\quad \text{for sequences of operations } g \text{ and } h \end{aligned}$$

4.6 Scenario-based resource prediction

In the Robocop component model, multiple components can work on different tasks that are executed in parallel. A scenario is defined as a set of tasks. This set contains the tasks that are executed in parallel, and for which the resource estimation needs to be made.

We can now define the weight function W for a scenario S and resource r, consisting of multiple tasks. This function computes for each task the estimated resource consumption:

$$\begin{aligned} W_s(\{t\}, r) &= W_t(t, r) && (13)\\ W_s(\{t_1, \ldots, t_n\}, r) &= W_s(\{t_1\}, r) \otimes \ldots \otimes W_s(\{t_n\}, r) && (14)\\ W(S, r) &= TS(W_s(S, r)) && (15) \end{aligned}$$

The function W introduces the functions W_s and TS and a new resource combinator $\otimes$. This resource combinator is used to combine resources that are consumed in parallel. The resources can be combined only after we know what type of scheduling is used. Therefore the introduced resource combinator $\otimes$ is abstract. It is the task of the function TS to do the scheduling and transform the expression in an expression which only contains concrete operators.

In the next section we will demonstrate the use of the weight functions w, Wop, Wt, Ws, and W in an example.

5 Example

In this section we will discuss a small example to illustrate how the method works. We will present a design of a media player build out of 9 components, a FileReader, a Decoder, a AudioDecoder, a VideoDecoder, a AudioRenderer, a VideoRenderer and 3 Buffer components.

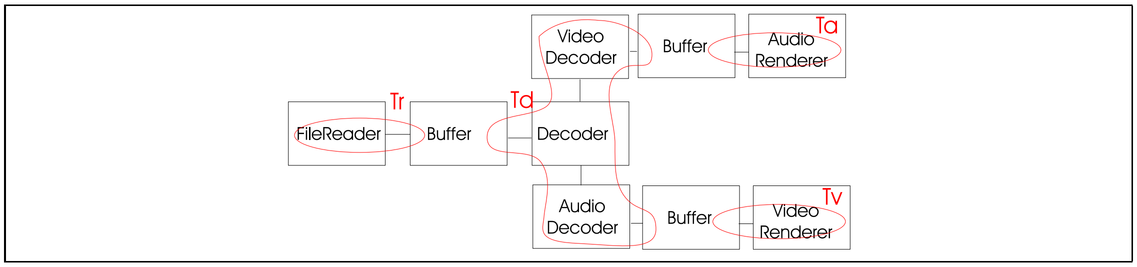

Fig. 7. Mediaplayer.

The media player application has 4 tasks. A task (Tr) to do the reading from file, one (Td) to decode the video and audio data, one (Ta) to render audio and one (Tv) to render video. The system described above is illustrated by figure 7. The service definitions can be found in figure 8.

In our example we consider a mediaplayer running multiple continuous tasks. We consider the scenario illustrated in figure 7 and are interested in the resource cpu.

$$\begin{aligned} S &= \{Tr, Td, Ta, Tv\} \\ r &= cpu \end{aligned}$$

We want to predict the cpu usage of scenario S.

$$\begin{aligned} W(S, cpu) &= TS(W_s(S, cpu)) \\ &= TS(W_s(\{Tr, Td, Ta, Tv\}, cpu)) \\ &= TS(W_t(Tr, cpu) \otimes W_t(Td, cpu) \otimes W_t(Ta, cpu) \otimes W_t(Tv, cpu)) \end{aligned}$$

Now we are going to use the task definitions and the scheduling function. We consider Round Robin scheduling. This means that if a task A requires 100 cycles for a loop and there are n tasks, A will have performed a loop each n * 100 cycles. The tasks and the scheduling function are defined as follows:

$$\begin{aligned} TS(A_1 \otimes \ldots \otimes A_n) &= (B_1, \ldots, B_n) \\ &\qquad \text{where } B_i = A_i * n \end{aligned}$$

$$\begin{aligned} Tr &= IFileReader.Read \\ Td &= IDecode.Decode \\ Ta &= IAudioRenderer.Render \\ Tv &= IVideoRenderer.Render \end{aligned}$$

These definitions enable us to evaluate $W(S, r)$ further. We now have the following intermediate result:

$$\begin{aligned} W(S, cpu) = (&W_{op}(IFileReader.Read, cpu) * 4, \\ &W_{op}(IDecode.Decode, cpu) * 4, \\ &W_{op}(IAudioRenderer.Render, cpu) * 4, \\ &W_{op}(IVideoRenderer.Render, cpu) * 4) \end{aligned}$$

```
FileReader
  requires IBuffer
  provides IFileReader {
    operation Read
      resource cpu
        require 1000
      behavior
      operation Read calls:
        IBuffer.Put
  }
```

```
AudioDecoder
  requires IBuffer
  provides IAudioDecoder{
    operation Decode
      resource cpu
        require 5000
      behavior
      operation Decode calls:
        IBuffer.Put*
  }
```

```
AudioRenderer
  requires IBuffer
  provides IAudioRenderer{
    operation Render
      resource cpu
        require 200
      behavior
      operation Decode calls:
        IBuffer.Get
  }
```

```
Buffer
  provides IBuffer{
    operation Put
      resource cpu
        require 50
      behaviour
      operation Put calls:
    operation Get
      resource cpu
        require 50
      behavior
      operation Get calls:
  }
```

```
Decoder
 requires IBuffer
          IAudioDecoder
          IVideoDecoder
 provides IDecoder{
      operation Decode
      resource cpu
        require 100
      behavior
      operation Decode calls:
        IBuffer.Get;
        IAudioDecoder.Decode;
        IVideoDecoder.Decode
 }
```

```
VideoDecoder
  requires IBuffer
  provides IVideoDecoder{
    operation Decode
      resource cpu
        require 20000
      behavior
      operation Decode calls:
        IBuffer.Put*
  }
```

```
VideoRenderer
  provides IVideoRenderer{
    operation Render
      resource cpu
        require 1200
      behavior
      operation Decode calls:
        IBuffer.Get
  }
```

Fig. 8. Service specifications

We will now show how $W_{op}(IDecode.Decode, cpu)$ is evaluated. This means a set of services that implement $IDecode.Decode$ will be generated, the behavior of all operations called for the implementation of $IDecode.Decode$ will be generated. We will use the following abbreviations $D = Decoder$, $Vd = VideoDecoder$, $Ad = AudioDecoder$ and $B = Buffer$.

$$\begin{aligned} W_{op}(IDecode.Decode, cpu) &= w(IDecode.Decode(C(IDecode.Decode)), cpu) \\ &= w(IDecode.Decode(D, Vd, Ad, B), cpu) \\ &= w(IDecode.Decode(D), cpu) \oslash \\ &\left(\begin{array}{c} w(IBuffer.Get(Vd, Ad, B), cpu) \\ \oplus \\ w(IAudioDecoder.Decode(Vd, Ad, B), cpu) \\ \oplus \\ w(IVideoDecoder.Decode(Vd, Ad, B), cpu) \end{array} \right) \end{aligned}$$

To continue the example we show how $w(IBuffer.Get(Vd, Ad, B), cpu)$ evaluates in the next step.

$$\begin{aligned} w(IBuffer.Get(Vd, Ad, B), cpu) &= w(IBuffer.Get(B), cpu) \\ &= 50 \end{aligned}$$

After we evaluate the other w functions we get the following result.

$$\begin{aligned} W_{op}(IDecode.Decode, cpu) &= 100 \oslash \\ &\left(\begin{array}{c} 50 \\ \oplus \\ (5000 \oslash (\lambda l \rightarrow l \times 50)) \\ \oplus \\ (20000 \oslash (\lambda k \rightarrow k \times 50)) \end{array} \right) \end{aligned}$$

Before we can continue evaluating this result we need to instantiate the two lambda expressions. We assume decoding one data packet results in 10 packets for the audio buffer and 5 for the video buffer. Therefore we instantiate the variable l representing the number of Puts in the audiobuffer with 10 and the variable k representing the number of Puts in the videobuffer with 5.

The last thing we need for the evaluation of $W_{op}(IDecode.Decode, cpu)$ is the definitions of the resource combinators. During our calculations we use the following resource consumption operator definitions:

$$\begin{aligned} a \oplus b &= a + b \\ a \oslash b &= a + b \end{aligned}$$

At this time we are able to evaluate $W_{op}(IDecode.Decode, cpu)$ completely.

$$W_{op}(IDecode.Decode, cpu) = 25900$$

After evaluating the other W_{op} functions we get the required cpu cycles required for the execution of one loop for the individual tasks in scenario S:

$$W(S, cpu) = (1050 * 4, 25900 * 4, 250 * 4, 1250 * 4)$$

6 Concluding remarks

6.1 Discussion

Until now, we applied our prediction method only to a small application. However, within the Robocop project, we have access to real-world applications. Therefore, we are at present applying our method to industrial software systems of some of our partners. The experiments that we have done thus far are very promising.

Our prediction technique has the following advantages:

- The method fits very well with current design approaches for reactive systems such as UML [8] or [9] where the dynamics of systems are modeled using scenarioss. Hence, the effort needed for modeling the resource use in addition to an existing specification is limited. Furthermore, the modeling of scenarioss typically happens early in the development process. Hence, our approach can be used to obtain feedback during the early stages of design.
- The method is compositional: results about the resource of a system can be computed based on specifications that are provided for its constituent components.
- The method is general. It can be applied to different types of resources and for different types of estimations. This is achieved by allowing different definitions of resource combinators.

The approach also has some limitations that need further research:

- We assume that consumption is constant per operation, whereas it typically depends on parameters passed to operations and previous actions. Extending the method in this respect is future work;
- Our scenario-based approach depends on the ability to define realistic runs. This makes our approach dependent on the quality of an architect;
- The model presented does not deal with synchronization between tasks.

6.2 Related work

The development of methods and techniques for the prediction of properties of component based systems is a notoriously hard problem and has been the subject of recent OOPSLA and ICSE workshops. A guideline for research projects in predictable assembly is proposed in [4].

An example approach that addresses the prediction of end-to-end latency is presented in [10]. Based on our experience, we can subscribe the statement of Hissam et. al. that it is necessary that prediction and analysis technology are part of a component technology in order to enable reliable assembly of component-based systems. However, there are many alternative ways in which the two may be technically related across the different phases of the component life cycle.

Prior work of an predictable assembly approach for estimating static resource use (using a Koala-like component model) is described in [7]. This builds on earlier work on prediction of static memory consumption described in [6].

The area of software architecture evaluation is dominated by scenario-based approaches [3, 12]. The disclaimer of these approaches is that the quality of their results depends on the representativeness of the scenarios chosen. The same disclaimer applies to the approach we propose in this paper.

The main differences of our approach are that we use a formal model: our method addresses dynamic instead of static resource consumption, supports resource consumption analysis in the early phases of software development and prediction requires little effort.

6.3 Contributions

In a previous paper [5] we presented a first approach for predicting resource use of component-based systems. This paper extends that approach such that it is able to deal with concurrent tasks that share system resources. To support this, we extended our approach with a means of mapping logical scenarios to tasks. Subsequently, these tasks are subject to system-wide resource scheduling policies. We have shown how these *global* scheduling policies can be taken into account into the *component-based* prediction technique. We conclude that for doing predictions for multitask systems, it does not suffice to have only component-level information, but that also system-wide resource sharing policies need to be taken into account.

In the previous paper we focused on memory-use. In this paper we focused on CPU-use. This led to the need to classify resources into different categories (pre-emptive versus non-pre-emptive and processing versus non-processing). For each category, different formulae are needed for combining different types of resources.

We have illustrated our approach to a case study that is closely inspired by an actual industrial system.

6.4 Acknowledgments

We would like to thank Reinder Bril, Merijn de Jonge, and Rudolf Mak for their comments.

References

1. D. Box. *Essential COM*. Object Technology Series. Addison-Wesley, 1997.
2. E. Clarke, O. Grumber, and D. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.
3. P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures*. SEI Series in Software Engineering. Addison-Wesley, 2002.
4. I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau. Anatomy of a research project in predictable assembly. In *5th ICSE Workshop on Component-Based Software Engineering*. ACM, May 2002.
5. M. de Jonge, J. Muskens, and M. Chaudron. Scenario-based prediction of run-time resource consumption in component-based software systems. In *Proceedings: 6th ICSE Workshop on Component Based Software Engineering: Automated Reasoning and Prediction*. ACM, June 2003.
6. E. Eskenazi, A. Fioukov, D. Hammer, and M. Chaudron. Estimation of static memory consumption for systems built from source code components. In *9th IEEE Conference and Workshops on Engineering of Computer-Based Systems*. IEEE Computer Society Press, Apr. 2002.
7. A. V. Fioukov, E. M. Eskenazi, D. Hammer, and M. Chaudron. Evaluation of static properties for component-based architectures. In *Proceedings of 28th EUROMICRO conference, Component-based Software Engineering track*. IEEE Computer Society Press, Sept. 2002.
8. M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley, 2003.
9. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, 2003.
10. S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Packaging predictable assembly. In *Proceedings of First International IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2002.
11. International Telecommunications Union. *ITU-TS Recommendation Z.120: Message Sequence Charts (MSC)*, 1996.
12. R. Kazman, S. Carriere, and S. Woods. Toward a discipline of scenario-based architectural engineering. *Annals of Software Engineering*, 9:5–33, 2000.
13. T. Mowbray and R. Zahavi. *Essential Corba*. John Wiley and Sons, 1995.
14. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, Mar. 2000.