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

Merijn de Jonge
M.de.Jonge@tue.nl

Johan Muskens
J.Muskens@tue.nl

Michel Chaudron
M.R.V.Chaudron@tue.nl

Department of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

Resources of embedded systems, such as memory size and CPU power, are expensive and (usually) not extensible during the lifetime of a system. It is therefore desirable to be able to determine the resource consumption of an application as early as possible in the design phase. Only then, a designer is able to guarantee that an application will fit on a target device.

Resource prediction is a technique to estimate the amount of consumed resources by analyzing the design and/or implementation of an application. In this paper we concentrate on predicting memory consumption in component-based applications. Component-based applications complicate resource predictions because resource consumption is spread across individual components.

The challenge is to express resource consumption per component, and to combine them to do predictions over compositions of components. To that end, we propose a model in which individual resource estimations of components can be combined. These composed resource estimations are then used in scenarios (which model run-time behavior) to predict memory consumption of applications.

1. Introduction

In this paper we describe a method for estimating resource properties of component-based systems, using models of behavior. We model behavior per component in terms of message sequence charts [7]. Behavior of applications (i.e., compositions of components) is estimated by composing the behavior models of the individual components.

We use message sequence charts in our method because they are both intuitive for use in practical engineering projects, and sufficiently formal for automated analysis.

We propose a scenario-based prediction method in order to avoid the complexity of full state-space analysis of systems, as is encountered by model-checking approaches. The idea behind this is that 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 evaluation software architectures [8], the reliability/usability of the results of our analysis method depends on the representativeness of the scenario's that are used.

Our research was carried out in the context of Robocop.[1] The aim of Robocop is to define an open, component-based framework for the middle-ware 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 have limited resources (CPU, memory, power, etc.) and the applications they support typically have real-time requirements.

Resources in embedded systems are relatively expensive and (usually) cannot be extended during the lifetime of an appliance. 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, Robocop applications are subject to change. For example, applications can be replaced by newer versions or completely removed. Additionally, new applications can be downloaded on a device.

To make sure that an application fits on a particular target appliance, in combination with several other applications, knowledge about its resource consumption is required. Getting this information soon (preferably at design-time), can help to prevent resource conflicts at run-time. This would decrease development time and, consequently, leads to a reduction of development costs. In this paper we propose an approach for predicting memory consumption of applications. A key feature of this approach is that it can be used anywhere during the design and implementation.

[1] Robocop 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.

The paper is structured as follows. In Section 2 we discuss the Robocop component model that we use for our prediction method. In Section 3, we introduce scenario-based resource prediction. In Section 4 we show how to apply our method in practice. Section 5 addresses related work and draws some conclusions.

2. The Robocop component model

The Robocop component model is a variant of the Koala component model [9]. A Robocop component is a set of models, each of which 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 functional model, the non-functional model (modeling timeliness, reliability, memory use, etc.), the behavior model, and the simulation model.

The functionality offered by a component is logically modeled as a set of 'services'. 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 component may define several interfaces. The Robocop component model distinguishes 'provides' interfaces and 'requires' interfaces. The first defines the operations offered by the component. The latter defines the operations of other interfaces that the component uses.

In Robocop, as well as in other component models, component interfaces and component implementations are separated to support 'plug-compatibility'. This allows different components, implementing the same interfaces, to be replaced at any time. As a consequence, the actual implementation (service) to which a component is bound, is not known at the time of designing a component. This implies that resource consumption cannot be completely determined for an operation, until an application defines a concrete binding of services.

3. Resource prediction method

We want to determine the resource usage of component-based software systems. This run-time resource use depends on the way that components cooperate to realize operations and on the specific executions of the system.

In this section we describe an approach for predicting the resources that a system uses for a specific execution sequence. Tool support can easily be developed for this approach to automate prediction of resource consumption.

Service specifications Our approach considers composition of services. Each service has one or more provides interfaces and one or more requires interfaces. A provides interface lists the operation 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 operation in the provides interface, the specification lists the resources that are claimed before execution, as well as the resources that are released when execution terminates;
- *Uses*: Per operation a list of operations that are used from other interfaces;
- *Behavior*: For each operation in a provides interface, the sequence of operations it uses.

Figure 1 contains an example service specification. It is a specification for service s_1 which uses 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 claims 100 bytes of memory and on return, releases 100 bytes.[2] The behavior section of the service specification defines that operation g from interface I_2 is called *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 consumption, will be discussed shortly.

Resource consumption in our approach, is specified per operation. Therefore, resources claimed during service instantiation and resources released at service destruction are specified as part of constructor and destructor operations.

The sequence of operations from other interfaces called by an operation forms a message sequence chart (MSC) for that operation. Observe that these are partial sequences, because indirect operation calls (for instance operations that are called by $I_3.h$ in Figure 1) are not specified in the service specification.

The service specifications can be defined at design time, or partly be generated from an implementation. In the latter case, *call graph extraction* can be used to obtain MSCs and

[2] In this paper we focus on prediction of run-time memory usage. Prediction of other resource usage (such as CPU usage) is future work. Therefore we specify resources in terms of bytes of memory.

```
service s1
 requires I2
 requires I3

 provides I1 {
  operation f
   uses I2.g
   uses I3.h
   resource claim   100
            release 100
  behavior
  operation f calls:
    I2.g*;
    I3.h}
```

Figure 1. An example service specification.

to determine requires interfaces. In case a service specification was defined at design-time, service specification generation can serve to validate an implementation (i.e., to check whether a service specification, extracted from the implementation, corresponds to the one defined at design-time)

Composition of services As can be seen from Figure 1, a service 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 interface implementation (service). The application developer will decide which services to use for each required interface.

Two services may have different requires interfaces, even when both implement the same provides interface. This means that the exact sequence of operation calls is not known before an application is assembled. For resource estimation this implies that the resources are not only determined by the operations that are called from an application, but also from the concrete composition of services that constitute the application.

To assemble an application, we have to transitively determine and instantiate the required interfaces. Finding the required services transitively is defined by the composition function C. Given a set of services, service specifications, and an interface that is required, this function yields a composition of needed services. The function is defined as:

$$C(I_1 \circ \ldots \circ I_n) = C(I_1) \circ \ldots \circ C(I_n) \quad (1)$$

$$C(I) = s \circ C(I') \text{ for a service } s \quad (2)$$

where $I \in s.\textit{provides} \wedge I' = \cup s.\textit{uses.interface}$

After a composition is determined, MSCs are composed from partial MSCs of individual operations (see Figure 2).

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

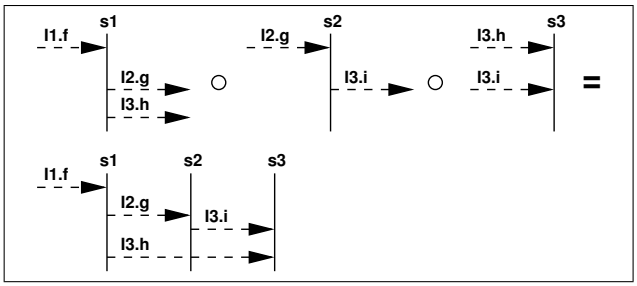

Figure 2. Composition of per-operation message sequence charts.

- The 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).
- All possible configurations are generated. This can be used for performance comparison in the next phase.

Per-operation resource measurement After a composition of services is determined with the composition function C, the resources of used operations can be predicted. To that end, we define a *weight* function w using 'resource combinators'. This function calculates resource consumption of an operation f over a composition of services, by accumulating the resources of f itself, *and* of the resource of all the operations that are called by f. This function is defined as:

$$w(f(s)) = (a, b) \quad (3)$$

where $a = s.f.\textit{resource.claim} \wedge b = s.f.\textit{resource.release}$

$$w(f(s_1 \circ \ldots \circ s_n)) = \begin{pmatrix} w(f(s_i)) \oslash \\ w(f_1(s_1 \circ \ldots \circ s_n \backslash s_i)) \\ \oplus \\ \vdots \\ \oplus \\ w(f_k(s_1 \circ \ldots \circ s_n \backslash s_i)) \end{pmatrix} \quad (4)$$

where s_i provides $f \wedge f_1 \ldots f_k \in s_i.f.calls$

The resource combinator $\oplus$ is used to model resource usage of two sequentially executed operations. The combinator $\oslash$ is used to model resource usage within an operator. That is, it serves to combine resource tuples (c_1, r_1) and (c_2, r_2) of operations f and g if operation g is called by operation f. The definitions of these combinators may be varied, in order

to measure resource usage in different ways. Section 4 gives a typical definition of these combinators.

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

$$W_{\mathsf{op}}(f(I)) \quad = \quad w(f(C(I))) \tag{5}$$

for interface I and operation f

This functions estimates the resource consumption of an operation call f from interface I. The function first determines a composition of services that (together) forms the 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.

Modeling resource consumption for call sequences Each operation call that is specified in a service specification may be accompanied with an iterator ('$\star$') to indicate that the operator is called a varying number of times. For the purpose of resource estimation, we represent a sequenced operator call as a lambda expression:

$$f\star \quad = \quad \lambda l \rightarrow l \times f \tag{6}$$

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

$$w(\lambda l \rightarrow l \times f(s)) \quad = \quad \lambda l \rightarrow l \times w(f(s)) \tag{7}$$

Such lambda expressions are instantiated with concrete numbers, when resource consumption for a complete application is predicted (see below).

Scenario-based resource measurement Now that we are able to estimate resource consumption for individual operations calls, we need to predict resource consumption for a complete application. To that end, we need to know what plausible sequences of operation calls are. Such sequences (called 'scenarios'), are determined together with implementors of services. If iterated operations are called in a scenario, the scenario becomes a lambda expression. Defining a scenario therefore consists of the following two steps:

1. Defining a plausible sequence of operation calls;
2. Instantiating the lambda expressions.

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

In the Robocop component model, multiple components can execute in parallel. Hence, two operations can either be called sequentially, or in parallel. We use the resource combinator ';' to denote sequential composition of operation calls, and the combinator '$\|$' for parallel composition.

We can now define the weight function W_{s} for a scenario, consisting of sequential and parallel operation calls. This function computes for each operation call the estimated resource consumption:

$$W_{\mathsf{s}}(f) \quad = \quad W_{\mathsf{op}}(f) \tag{8}$$
$$W_{\mathsf{s}}(f;g) \quad = \quad W_{\mathsf{s}}(f) \oplus W_{\mathsf{s}}(g) \tag{9}$$
$$W_{\mathsf{s}}(f \parallel g) \quad = \quad W_{\mathsf{s}}(f) \otimes W_{\mathsf{s}}(g) \tag{10}$$

This function defines how to measure resource consumption for operations that are called sequentially (using the resource combinator $\oplus$), and for operations that are called in parallel (using the resource combinator $\otimes$). Section 4 gives typical definitions for these resource combinators. Observe that this weight function may indicate a possible memory leak, in case it yields a tuple (c, r) where $c > r$.

By applying W_{s} to different scenarios, the reliability of predictions will improve. Different types of scenarios give information about typical uses of an application. For instance, a 'worse-case' scenario, gives information about maximal resource consumption; a 'mean-use' scenario, about consumption for ordinary use; and a 'maximum-performance' scenario gives resource consumption in the case that performance should be maximized.

4. A case study

In this section we discuss a prediction experiment for a chat application. The application is a composition of 4 services (`MessageProducer`, `MessageConsumer`, `ClntSocket`, and `SrvSocket`). The service specifications for the first two components are depicted in Figure 3 and 4.[3]

According to these specifications, the chat application requires a composition of services that implements the interfaces `IMessageConsumer` and `IMessageProducer`:

$$C(\texttt{IMessageConsumer} \circ \texttt{IMessageProducer})$$

A typical scenario S for the chat application, consisting of sequential and parallel operation calls, might be defined as:[4]

$$\begin{aligned} S \quad = \quad & \texttt{MessageConsumer}; \texttt{MessageProducer}; \\ & (\texttt{Produce} \parallel \texttt{Consume}); \\ & \sim\texttt{MessageConsumer}; \sim\texttt{MessageProducer} \end{aligned} \tag{11}$$

From the service specifications we see that the operators Produce and Consume are iterated operators. They can

[3] Due to space limitations, the service specifications for `IClntSocket` and `ISrvSocket` are not shown. The components that implement these interfaces consume 24 and 48 bytes of memory during service instantiation, respectively.

[4] For readability we omitted the interface names for each operation call.

```
service MessageProducer
  requires IClntSocket
  provides IMessageProducer {
  operation MessageProducer:
    uses {}
    resource claim   1 release 0
  operation ~MessageProducer:
    uses {}
    resource claim   0 release 1
  operation Produce
    uses: IClntSocket.ClntSocket
          IClntSocket.~ClntSocket
          IClntSocket.Send
    resource claim   0 release 0
  behavior
    operation MessageProducer calls:{}
    operation ~MessageProducer calls:{}
    operation Produce calls:
        (IClntSocket.ClntSocket;
        IClntSocket.Send;
        IClntSocket.~ClntSocket)*}
```

Figure 3. MessageProducer service spec.

```
service MessageConsumer
  requires ISrvSocket
  provides IMessageConsumer {
  operation MessageConsumer:
    uses {}
    resource claim   56 release 0
  operation ~MessageConsumer:
    uses {}
    resource claim   0 release 56
  operation Consume
    uses:ISrvSocket.ISrvSocket
         ISrvSocket.~ISrvSocket
         ISrvSocket.Receive
    resource claim   8208 release 8208
  behavior
    operation MessageConsumer calls: {}
    operation ~MessageConsumer calls: {}
    operation Consume calls:
        (ISrvSocket.SrvSocket;
        ISrvSocket.Receive;
        ISrvSocket.~SrvSocket)*}
```

Figure 4. MessageConsumer service spec.

therefore be represented as lambda expressions:

$$\lambda l_1 \rightarrow l_1 \times (\texttt{ClntSocket}; \texttt{Send}; \sim\texttt{ClntSocket})$$
$$\lambda l_2 \rightarrow l_2 \times (\texttt{SrvSocket}; \texttt{Receive}; \sim\texttt{SrvSocket})$$

The lambda variables l_1 and l_2 correspond to the number of messages sent and the number of messages received, respectively. Because of these lambda expressions, the scenario S is also a lambda expression, and so is the corresponding weight function for S:

$$\lambda l_1 l_2 \rightarrow W_{senario}(S)$$

In order to evaluate the weight function for S, the lambda expression needs to be instantiated. We performed an experiment in which we compared the prediction for sending and receiving 2 messages with the measurements on the real application. Therefore we instantiated both variables with 2, corresponding to the number of messages transfered.

For the experiment we used the following definitions for the resource combinators $\oslash$, $\oplus$, and $\otimes$ in order to estimate the number of bytes allocated on the heap:

$$\begin{aligned}(a,b) \oplus (c,d) &= (a-b+c,d) \text{ if } c \geq b \quad (12)\\ &= (a,b+d-c) \text{ if } c < b\\ (a,b) \otimes (c,d) &= (a+c,b+d) \quad (13)\\ (a,b) \oslash (c,d) &= (a+c,b+d) \quad (14)\end{aligned}$$

Evaluation of $\lambda l_1, l_2 \rightarrow W_s(S)$ using these definitions, and instantiating the lambda variables with 2, yields the tuple $(8337, 8337)$. The function W_s thus predicts a worse-case memory consumption of 8337 bytes. The analysis did not signal a memory leak, because the amount of memory that is used by the application is also released. Analysis of the execution of the chat application indicates a consumption of 9716 bytes. This corresponds to 8337 bytes used by the services, and 1280 bytes of overhead. We see that the prediction of resource consumption by the services matches the analysis of the running application.

5. Concluding remarks

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 more realistic software systems of some of our industrial partners. The experiments that we did thus far are very promising:

- The method uses information that is commonly modeled in the design of software systems. Thus, the additional effort needed to apply our method is limited;
- Our method is compositional. Information about resource consumption per component is reused to predict resource consumption for component compositions;
- Prediction is simplified because knowledge of system architects is used to define critical scenarios, which would otherwise be defined on a trial and error basis;
- The method can be used early in the design. Since estimations can be made for incomplete parts of a system,

the design does not need to be completely modeled;

- Different definitions of the resource combinators, which combine the resource consumption of operations, can be given. This allows different ways of modeling resource consumption. For instance, in Section 4 we measured maximal memory consumption. Other definitions would provide information about memory usage over time, mean memory usage, etc.

Despite these promising advantages, our model also has some drawbacks/limitations that need further research. First, we assume that resource claims and releases are constant per operation, whereas it typically depends on parameters passed to operations. We do not yet know how to model this. Second, our scenario-based approach depends on the ability to define realistic runs. This makes our approach dependent on the availability of an architect. Furthermore, it has the potential danger that we fail to correctly locate the most critical (resource consuming) runs. Third, instantiating lambda expressions in case of iterated operation calls is difficult. It requires good insight in the implementation of services. Fourth, we cannot yet model different memory allocation strategies, which is required for doing realistic estimations. Addressing these issues is future work.

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 several OOPSLA and ICSE workshops. A guideline for research projects in predictable assembly is proposed in [10]. A much needed proposal for standard terminology in the domain can be found in [1].

An example approach that addresses the prediction of end-to-end latency is presented in [6]. Based on our experience, we 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.

MSCs are standardized in [7] and have a formal semantics (see e.g., [4]). MSCs are used for different types of analysis. Examples are the analysis of logical system properties (such as liveness and deadlock) and the analysis of performance and timeliness properties. We are aware of no other work that uses MSCs for estimating run-time memory use. Prior work of an predictable assembly approach for estimating static resource use (using a Koala-like component model) is described in [5]. This builds on earlier work on prediction of static memory consumption described in [3].

The area of software architecture evaluation is dominated by scenario-based approaches [2, 8]. The disclaimer of these approaches is that the quality of their results depends on the representativeness of the scenario's chosen. This also holds for the approach we propose in this paper.

Contributions In this paper we addressed prediction of run-time memory consumption in component-based applications. We proposed a formal model for modeling memory consumption per service operation and per application. Prediction is based on scenarios which model run-time behavior of applications. Our model supports i) multiple implementations per interface; ii) different ways of measuring resource consumption using resource combinators; iii) modeling of variable sequences of operation calls using lambda expressions; iv) per-component specification of resource consumption; v) composition of resource consumption specifications based on composition of MSCs.

References

[1] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, S. R., and K. Wallnau. Technical concepts of component-based software engineering, volume II. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[2] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectur*. SEI Series in Software Engineering. Addison-Wesley, 2002.

[3] 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.

[4] B. Finkbeiner and I. Kruger. Using message sequence charts for component-based formal verification. In *Proceedings of the OOPSLA workshop on Specification and Verification of Component-Based Systems*. ACM, Oct. 2001.

[5] 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.

[6] 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 *LNCS*. Springer-Verlag, June 2002.

[7] International Telecommunications Union. *ITU-TS Recommendation Z.120: Message Sequence Charts (MSC)*, 1996.

[8] R. Kazman, S. Carriere, and S. Woods. Toward a discipline of scenario-based architectural engineering. *Annals of Software Engineering*, 9:5–33, 2000.

[9] 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.

[10] A. Wall, M. Larsson, C. Norström, and I. Crnkovic. Using prediction enabled technologies for embedded product line architectures. In *5th ICSE Workshop on Component-Based Software Engineering*. ACM, May 2002.