Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 114 (2005) 25–45
www.elsevier.com/locate/entcs

Estimating Non-functional Properties of Component-based Software Based on Resource Consumption

Marcus Meyerhöfer[a,2,1], Klaus Meyer-Wegener[a,3]

[a] *Department of Computer Science 6*
Friedrich-Alexander University of Erlangen and Nuremberg
Erlangen, Germany

Abstract

The consideration of non-functional properties in the component-oriented approach to software development is important for its success. The COMQUAD project defines a system architecture and a development methodology for component-based software with quantitative properties and adaptivity, thereby respecting non-functional properties from design to provision at runtime. Based on an elementary model of software components, we show how response time as an exemplary property is treated: Its relation to available resources is investigated and it is shown how resource requirements of the whole system can be derived from knowledge about the constituent components.

Keywords: Non-functional properties, software components, estimation, resource consumption, assembly matrix, response time.

1 Introduction

A component-oriented approach to software development allows to construct software from already existing, well-tested and documented parts, and therefore fosters reuse. While there are many different definitions of a component, none is commonly agreed upon. Here, the definition of Szyperski in [22] is used: "A software component is a unit of composition with contractually specified

[1] This work is supported by DFG grant FOR 428.
[2] Email: Marcus.Meyerhoefer@uni-erlangen.de
[3] Email: Klaus.Meyer-Wegener@uni-erlangen.de

doi:10.1016/j.entcs.2004.02.066

interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Therefore, software components are expected to be suitable units of reuse that have no implicit dependencies and thus ease application development.

While the functionality is the most important factor in selecting a component for an application that is being developed, non-functional properties (NFPs) are becoming more and more important, too. These NFPs should not be determined afterwards, but must be considered in the design process from the beginning and thus influence the selection of components. The designer should be able to estimate the NFPs of the system being built at any point in time.

The COMQUAD project [4] defines a system architecture and a development methodology for component-based software with quantitative properties and adaptivity. The specification of NFPs is used in the design phase to select components not only on the basis of an expected interface or function, but also on the basis of the NFPs. The runtime system is constructed in order to ensure that the system being build (the so-called "assembly") later actually has the specified NFPs. It is not detailed any further in this paper; for details see [6,14].

The term "non-functional property" can relate to many different things, e.g. performance, quality of service (being itself a set of properties), security, safety, robustness, portability, etc. In the context of component software, catalogs of such properties have already been published, e.g. [3]. They present a limited set of properties and categorize them. However, it seems to be impossible to treat all properties in a similar fashion [5]. Therefore, a few of these properties should be selected to start with, of course striving to extend the approach to more properties in the future. For the purpose of this paper, the well-known property of response time is selected, as its significance for most applications is commonly understood. Note however, that there are several ways to define response time (see [17]). In this paper, an elementary component model is introduced as a basis, focusing on imported and exported interfaces of a component and the methods contained in them. The invocation relation among exported and imported interfaces is represented in form of a matrix. Response time has the particular property of being additive, which

[4] "COMponents with QUantitative properties and ADaptivity" is a project funded by the German Research Council. It started October 1, 2001, at Dresden University of Technology and Friedrich-Alexander University of Erlangen and Nuremberg - see `http://www.comquad.org`.

[5] A reason could be that they differ strongly in their nature. Think of e.g. transactional behavior having a nominal scale in contrast to throughput or response time having a ratio scale.

means that the response time of a component is determined by the response times of all the other components it imports and by its own processing. The latter part is further determined by the available resources.

Resources that influence the response times are primarily memory and processor time. Given a certain availability of these resources on a platform, response times can be calculated. In the opposite direction, resource requirements could be derived from a given response-time requirement. Then the resources can be reserved, and appropriate scheduling algorithms can be applied.

These calculations can only be done if the expected resource usage of each component is known a priori, that is, its CPU and memory usage. This paper assumes a model that describes the resource usage of components. With that, the cumulative resource requirements of the whole system can be derived.

2 Related work

A couple of projects address similar problems. In an overview paper, [18] Sitaraman describes many fundamental problems that have to be considered when performance of component-based systems is to be specified and reasoned about. The specification must be "value-based", that is, it must be based on the values of objects involved, not just on their sizes[6]. The consequence is that performance specification needs a functional specification as its basis to determine the values of objects involved. The need for so-called point-wise specification giving precise statements based on input and output values is stated, because otherwise only very coarse worst-case estimates can be used. Regarding the question at which level of precision the performance should be specified, there is no general answer. The author proposes to specify performance at different levels, enabling users to select the most appropriate one for a given context. In general, the paper gives some hints on which directions to follow, but does not go into the details of the solutions.

The RESOLVE framework deals with many aspects of component-oriented software, including performance. In [19] the authors present their approach to the performance specification of software components. They do not constrain themselves to a specific component model, but indicate behavior in terms of runtime and memory consumption in the context of the functional specification, which is expressed in a dialect of the RESOLVE language [23]. Upper bounds for the execution time that depend on parameter values and types are defined using predicates like **Is_O**, which resembles the "big-O" notation. For

[6] Processing complex objects (e.g. trees) may take much longer than processing simple objects (e.g. strings) of the very same size.

storage, another predicate **Is_RO** is used which is more strict in expressing upper bounds. These predicates are in fact valued-based, as has been indicated above. Hence, the resource specifications approximate the bounds using predicates, but rely on a detailed functional specification that exhibits the concrete implementation. They must be supplied by the developer of a component and obviously need a white-box view of the component. Furthermore, how to obtain the base values used in the specifications (e.g. initialization of an entry in the stack example) is not clearly stated.

In an earlier paper, Sreerama et al. show how to implement component-based software engineering in an object-oriented context, using C++ templates as building blocks for performance tuning [21]. In this approach, layered components can be "plugged" together without source-code modification, using different implementations for the same functionality that exhibit different performance behaviors. Obviously, one prerequisite for this is a performance specification of the different implementations to guide the selection of the appropriate component. This specification used only the big-O notation and thus remained rather imprecise. Also, it is not yet value-based.

The RASC project [15] uses a contractual approach for resource-aware software components, associating a precise specification with each component and monitoring access and consumption at runtime [8]. Using an extension to the Java runtime environment, all resources are reified through objects which are either specific to the system (e.g. CPU) or just alternative implementations of the standard Java API (e.g. Socket). The notion of component in RASC is limited to standard Java programs or applets. To specify their resource usage, a specific abstract method must be implemented that returns so-called "resource utilization profiles." These profiles use a given set of classes and interfaces to model resource usage [20] and are read by resource monitors which are consulted by the reified resource objects to make sure that the component does not exceed specified usage. As the reification of resources like memory and especially CPU is a difficult task, they are modelled and monitored through information supplied by the `/proc` pseudo-filesystem of the underlying Linux operating system. Furthermore, these resources cannot be monitored synchronously as resources accessed via a reification object, but have to be polled asynchronously giving some kind of "status" report. To summarize, the specification is done at a rather low level, and it has to be provided by the implementer as part of a component. The expected resource usage is specified statically, it cannot refer to parameter values of the offered methods. Furthermore, it is not discussed how these values can be obtained, especially how CPU usage can be expressed independently of the properties of the runtime system.

The specification of components in the context of embedded appliances is investigated in [5]. It not only includes provided and required interfaces, but also the resources needed by each operation. Additionally, for each operation a list of the used operations from the required interfaces is provided together with a path expression that indicates the sequence of the calls. The resource usage is modelled at a coarse level, just specifying what is claimed by an operation before execution and what is released afterwards. The specification is again static and does not address parameters or return values of the operation. Using the specifications, the authors predict memory usage for a complete application, relying on the developer-supplied knowledge of plausible call sequences, called "scenarios." They turn the arbitrary number of calls in the path expressions ("*") into a particular number. It is not documented how the developer obtains those numbers. Also, CPU usage is left to further work. The goal is only to estimate total resource usage, response times of the system are not considered.

The COMPAS project [10] uses a matrix to describe the import-export relation of components. This matrix indicates for each pair of methods in the system the probability that method 1 calls method 2 in a particular transaction, that is, a particular workload. The information is extracted from traces. So the system must be complete before the evaluation can start. The purpose is to predict the behavior of the same system under new workloads from the measurements. But the development process itself is not modified to take care of performance issues while building the system.

So none of the approaches covers the full spectrum of including NFPs in the development process and the runtime support. Ideas from all the projects can however be used and combined in a new fashion.

3 Models

So the challenge still is to create models that integrate all the approaches presented so far and that go into more detail to allow for resource reservation and scheduling.

3.1 Component Model

For the purpose of this paper a quite general component model can be used. This model describes components mainly as software building blocks having declared interfaces of services a component offers and interfaces a component imports, that is the interfaces of other components it will call to implement its own functionality. An interface is a set of methods. We come from an Enterprise-Java-Beans-like [12] component model, focusing on the main prop-

erties and modeling the container as the runtime enviroment similar to a component. This is possible, as we do not model any of the middleware services the container offers but for the purpose of this model are just interested in inter-component and component-to-container method calls.

The assumed communication between components is synchronous method invocation; we do not cover e.g. multi-party communication or event models. Furthermore, we disallow cyclic call-graphs [7] .

Let C denote the set of all components c available for the system under construction, be it from a component repository, be it newly developed. Each software component c is a reusable unit of software, subject to deployment and composition, that offers a specific service described and accessible by a set of exported interfaces. A component itself may depend on other components, which is described as a set of imported interfaces. The set of interfaces exported or imported by any of these components is called $I = \{I_j | 1 \leq j \leq p\}$. A relation $R_{exp} \subseteq C \times I$ states the fact that a particular component c exports interface I_j: $(c, I_j) \in R_{exp}$. It is useful to have an associated predicate $exp(c, I_j) \equiv (c, I_j) \in R_{exp}$. Then $I_{c,exp} = \{I_j | exp(c, I_j)\}$ is the set of all interfaces exported by component c. Similarly, relation $R_{imp} \subseteq C \times I$ describing that c imports interface I_j: $(c, I_j) \in R_{imp} \equiv imp(c, I_j)$ and $I_{c,imp} = \{I_j | imp(c, I_j)\}$. Note that $I_{c,exp}$ must not be empty, but $I_{c,imp}$ can be, for component c having no dependencies. Then we define

$$\begin{aligned} I_{exp} &:= \{I_j | \exists c \in C : exp(c, I_j)\} \\ &= \bigcup_{c \in C} I_{c,exp} \end{aligned}$$

as the set of all exported interfaces and

$$\begin{aligned} I_{imp} &:= \{I_j | \exists c \in C : imp(c, I_j)\} \\ &= \bigcup_{c \in C} I_{c,imp} \end{aligned}$$

as the set of all imported interfaces, with $I = I_{imp} \cup I_{exp}$. As an integrity constraint, we require that all imported interfaces must be exported by at least one component:

$$\forall I_j \in I_{imp} : \exists c \in C : exp(c, I_j)$$

or $I_{imp} \subseteq I_{exp}$. It is not forbidden, however, that a component imports an

[7] However, while this might be a drawback, recursive method implementation is still allowed as we only capture method invocations between component instances. We do not count component internal method calls and therefore a recursive implementation will appear as a method not calling other business methods of other components.

interface that it itself exports.

Each interface I_j consists of a set of methods $m_{j,k}$, that is $I_j = \{m_{j,k} | 1 \leq k \leq n_j\}$. Please note that this means that methods belong to exactly one interface. Interfaces of our component model do not share methods. So a method always identifies the interface it belongs to. For easier handling in the following [8], we collect all methods of the several interfaces exported or imported by the same component in two sets:

$$M_{c,exp} = \bigcup_{I_j \in I_{c,exp}} I_j$$
$$= \{m_k | 1 \leq k \leq q_{c,exp}\} \text{ with } q_{c,exp} = \sum_{I_j \in I_{c,exp}} n_j$$

is the set of all exported methods of a component c and

$$M_{c,imp} = \bigcup_{I_j \in I_{c,imp}} I_j$$
$$= \{m_k | 1 \leq k \leq q_{c,imp}\} \text{ with } q_{c,imp} = \sum_{I_j \in I_{c,imp}} n_j$$

the set of all imported methods [9] of a component c.

Then $M_{exp} = \bigcup_{c \in C} M_{c,exp}$ is the set of all exported, and $M_{imp} = \bigcup_{c \in C} M_{c,imp}$ the set of all imported methods of all components, respectively. Finally, $M := M_{imp} \cup M_{exp}$.

The interfaces a component offers are defined before the component exists, so they must be accepted as they are. The imported interfaces, however, are selected by the implementation of a component. In more detail, it decides which methods of the imported interfaces are called by any method of an exported interface. This decision is captured in form of a relation $R_{impl} \subseteq M_{exp} \times C \times M_{imp}$, with $(m_{j1,k1}, c, m_{j2,k2}) \in R_{impl}$ saying that method $m_{j1,k1}$ (of interface I_{j1}) is implemented in component c by calling method $m_{j2,k2}$ (of interface I_{j2}). R_{impl} can be seen as a call graph as known from interprocedural analysis algorithms like e.g. [11]. However, R_{impl} is not determined by static analysis but is determined by the explicit specification of imported components that has to accompany a software component. Static analysis can be done to derive R_{impl}

[8] In the following presentation, we mostly consider the component level; therefore we omit the index j of $m_{j,k}$ for ease of reading. Of course one should have in mind that methods are grouped into interfaces.

[9] Please note that this definition implies a one-to-many relationship between interfaces and methods. This appears to be sufficient to capture reality. Overlapping of interfaces does not offer any additional benefit, since it unnecessarily restricts the implementation of the interfaces.

if all implementations are at hand, but usually components are developed independently from each other and therefore an implementer decides only which interfaces to import from other components. The application assembler uses this information to provide an implementation for the interfaces imported. Of course, for R_{impl} it must hold that:

$I_{j1} \in I_{c,exp}$
$I_{j2} \in I_{c,imp}$

To have this information is crucial when addressing the NFPs of components. The relation at the moment does not show the dependence on the parameters of the methods. That would be even more useful, but it would make the relation much more complex. So we start with the elementary form with the clear intension to later extend it.

So for a given component c, each exported method $m_{k1} \in M_{c,exp}$ calls a subset $M_{c,imp,k1} \subseteq M_{c,imp}$ with

$$M_{c,imp,k1} := \{m_k \in M_{c,imp} | m_{k1} \in M_{c,exp} \wedge (m_{k1}, c, m_k) \in R_{impl}\}$$

With that, we can specify imported and exported interfaces differently:

$$I_{c,imp} = \{I_j | \exists m_k \in M_{c,exp} : (m_k, c, m_{k1}) \in R_{impl} \wedge m_{k1} \in I_j\}$$

and

$$I_{c,exp} = \{I_j | \exists m_k \in M_{c,exp} : m_k \in I_j\}$$

A simple example should clarify these concepts: Component `SquareRoot` in figure 1 offers just one service, namely the calculation of the square root of its input parameter (using Newton's method for square roots [9]). It imports an interface `Math` that offers calculation of mean and square. Component `BasicMath` is an example of a component offering that interface `Math`; that component itself does not need any other component.

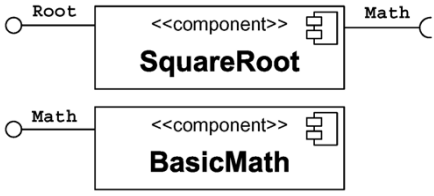

Fig. 1. Components `SquareRoot` and `BasicMath`

In IDL notation [13], the definition of components, interfaces, and methods would look like this:

```
interface Root {
   double squareRoot(double);
}

component SquareRoot {
   provides Root root;
   uses Math math;
}
```

```
interface Math {
   double square(double);
   double mean(double);
}

component BasicMath {
   provides Math math;
}
```

Using the notation introduced above, we have $C = \{SquareRoot, BasicMath\}$ and $M = \{square, mean, squareRoot\}$. Looking at component `SquareRoot`, we have

$$\begin{aligned} I_{SquareRoot,imp} &= \{Math\}, \\ M_{SquareRoot,imp} &= \{square, mean\}, \\ I_{SquareRoot,exp} &= \{Root\}, \\ M_{SquareRoot,exp} &= \{squareRoot\} \end{aligned}$$

The relation R_{impl} among the methods indicating which imported methods a exported method calls is defined as

$$R_{impl} = \{(squareRoot, SquareRoot, square), (squareRoot, SquareRoot, mean)\}$$

3.2 Non-functional Properties

It has just been said that the implementor of a component decides which interfaces to import. In our setting, components and their exported methods are subject to annotations regarding NFPs. These NFPs must be consistent with the NFPs of the components that offer the imported interfaces [19]. As mentioned above, we use response time as an example here.

Components always need resources when being executed. Many NFPs are also determined by those resources, and the relationship must be known if NFPs are to be derived. So this relationship must be identified and made part of the component specification.

For response time, the resources considered to be most important are CPU and memory [10]. In general, their usage depends on the platform and on the data processed. The data must be included if the relationship of resource usage and an NFP is to be identified. These data can be of a simple type like integer with its limited value range, or of more complex types like video or

[10] This assumes that the components are not distributed; otherwise the network would be the third kind of resource.

image. Value-based specification [18] may be feasible for the simple types, but for the complex types data values are much too large and too detailed, so the salient properties must be used instead, e.g. frame rate, compression format, and size in case of video. Both is denoted by *data* in the following, meaning either values or properties depending on the type.

So there is a function like:

$$NFP_{m_k} = f_{m_k}(resources, data)$$
$$rt_{m_k} = f_{m_k}(CPU, MEM, data)$$

with rt_{m_k} denoting the response time of method m_k and CPU standing for the CPU capacity actually used, irrespective of the reason—it can be just the available CPU, or it can be all the method m_k asked for. The same applies to MEM for the memory used. f_{m_k} denotes an arbitrary formula "calculating" the NFP of method m_k, depending on the resource usage and the data processed. By the way, state dependence of a component can be modelled as just another input parameter to f_{m_k} and thus is included in *data* already.

Such a formula could be supplied by the developer of the component, knowing its internal implementation and its relationship to the available resources. But even when the implementation is known, for complex components it cannot be expected that a developer is able to formulate such a formula precisely and correctly. So it is obvious that having such a function would be the ideal situation, but usually we may not able to derive a "sharp" function returning a fixed value for a given resource usage and data values or properties.

Taking a black-box view at a component or being – even as developer – not able to state a formula, the behavior of the component can still be measured under different conditions, that is with different input parameters and different resources. As the state space is usually much too large even for simple parameter types, an exhaustive measurement is not feasible. Instead, the state space can be split into regions and the NFPs are measured for each region. With additional knowledge, profiles of expected usage of common parameter values or properties, resp., can be stated and the component can be measured for these profiles only. Such an approach leads to a table of measured values, and therefore f_{m_k} is implemented as a table lookup (see table 1 with the example of response time).

Of course, these measurements are platform-dependent. To make them useful, they have to be transformed into numbers that are at least to some extent platform-independent. Details are still to be defined, however, the direction is the following: Given the processor speed and the CPU usage, the

Response time	Data	CPU	MEM
30ms	3,475,984	15 MIPS	100 KB
10ms	4,987	5 MIPS	50 KB
...	...	...	...

Table 1
Lookup table for resource usage, response time, and data

number of instructions executed by the method can be approximated [11]. So on a different platform with a different processor speed, the CPU usage can be estimated.

The table includes some simplifications. The values produced by measurements will usually not produce exact numbers, but intervals. It would be more appropriate to use these intervals as table entries. For the purpose of this presentation, we use only individual numbers.

For provision of resources, it is necessary to derive the resource usage of a component from a given NFP that this component should have. The data are either not specified, meaning that the NFP should be guaranteed for any data, or it can defined in terms of an exact value/property, a region, or a profile. This asks for inverse "functions" $f_{m_k}^{-1}$. Obviously, as the functions f_{m_k} are not bijective, the inverse functions yield sets of values for resource usage. In the case of the table lookup introduced above, the result is a relation for each m_k containing tuples (CPU, MEM) that state the resource usage for a given response time rt.

$$f_{m_k}^{-1}(NFP, data) = \{(resources_1), (resources_2), \ldots\}$$
$$f_{m_k}^{-1}(rt, data) = \{(CPU_1, MEM_1), (CPU_2, MEM_2), \ldots\}$$

Obviously, if the same response time results from different resource usages [12], the minimum for either resource can be chosen.

In the COMQUAD project, an extension of CQML [1] is used to describe the resource usages of NFPs (for details see [16,24]). In addition to the qualities and profiles of CQML, CQML+ allows to specify the resource demands a component has. Although in this paper we focus on CPU and memory usage, the extension is flexible enough to add arbitrary resource types, as the semantics are defined by the underlying resource manager [2].

[11] Assuming as a simplification that each CPU instruction has a constant execution time.
[12] Assume, for example, that a component uses caching. If more cache is available, the response time should decrease. However, at some point search in the cache plays such a significant role that it increases response time. So a smaller cache and a very large cache could indeed lead to the same response time.

Using again the simple example of `SquareRoot`, the following CQML+ code shows how annotations of NFPs could look like.

```
quality low_response_time (op: Operation)
{ response_time (op) < 2; }
quality med_response_time (op: Operation)
{ response_time (op) >= 5
  and response_time (op) < 7; }
quality high_error_bound (op: Operation)
  { square_error (op) < 0.000000001; }

quality memory_high (r: Resource)
{ size (r).minimum > 100; }
quality cpu_med (r: Resource)
{ instructions (r) = [100000,150000]; }

profile service_quality for SquareRoot {
  profile good {
    uses low_response_time (math.square);
    uses low_response_time (math.mean);
    provides med_response_time (root.squareRoot);
    provides high_error_bound (root.squareRoot);
    resources memory_high(memory);
    resources cpu_med(cpu);
    }
}
```

This definition requests a maximum response time and a specific error bound referring to the approximation. Of course, in order to be able to respond within the given time bound, `SquareRoot` must rely on the methods contained in the imported interface `math` of type `Math`, which themselves have to return their answers within a given (smaller) bound. Based on definitions of the quality characteristics `response_time` and `square_error` [13], three different qualities are defined, constraining `response_time` and `square_error`, respectively. Two further qualities constrain the resource usage using intervals. Finally, a profile associates those qualities with the component. As the component itself needs certain resources to offer medium response time and high error bound, this is stated in the resources clause of CQML+.

With such a specification, application designers can select a profile for

[13] These are actually user-defined types with a domain and optionally semantics coded in OCL—for brevity, they are omitted here. Examples can be found in [1,24].

`Square-Root` that fits their non-functional requirements. This then generates other requirements on imported interfaces, in this case on `Math`, regarding the response time of their methods. The next chapter will address the issue of plugging components together to form an assembly.

4 Assembly Structure

The previous chapter has focused on an individual component. This component shows some behavior which can be turned into resource demands. Such information should either be delivered by the developer of the component or has to be discovered by the user of the component through measurements. Furthermore, if the process and the parameters of measurement are known, a user (e.g. the application assember) can also check whether the specified properties are correct. Obviously, as the resource usage is platform specific it might be necessary anyway to measure the component for a new platform that the component vendor does not offer property descriptions for. Therefore, methods should be developed that enable the platform-independent description of such properties. The NFP's do not only depend on the resources, but also on the NFP's of other components which export the interfaces that the regarded component imports. So the composition (the assembly) of components must also be taken into account.

4.1 Establishing Relationships Among Components

To select the components needed for a particular system under construction (suc), the set of interfaces that this system is to export must be given: $I_{suc} \subseteq I$. We assume here that new interfaces have already been introduced by the design process and that components supplying these interfaces have been defined. They must be implemented anyway before any evaluation of non-functional properties can take place. The job of the designer (assembler) is then to select components that offer these interfaces. Selecting such a component may lead to new interfaces that must be supplied, namely those imported by the selected component. This process continues until no further interfaces are named by any of the selected components.

The designer creates component instances. The same component can be instantiated more than once in an assembly. This is not for load balancing or parallel processing, but if an interface is imported by several components, they should not be forced to use the same instance. In particular, both instances could have some different NFP's, because they can rely on different instances, too (see figure 2, where two possible assemblies are shown; assembly #1 uses two instances of component $c3$ having each a different subtree, while assembly

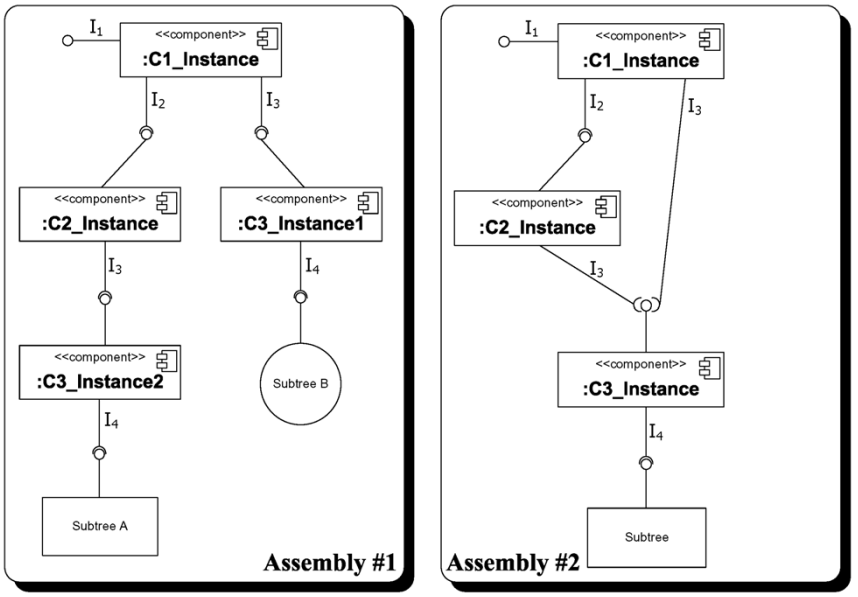

Fig. 2. two assemblies with different instances

#2 refers to one instance of $c3$, imported by two other instances). The set of component instances in an assembly is called CI. A function $f : CI \rightarrow C$ identifies the component for each instance. A relation $R_{assemb} \subseteq CI \times CI \times I$ states that an instance calls the other using an interface. The graph defined by the first two parts must be acyclic. Tupel $(ci_1, ci_2, I_1) \in R_{assemb}$ requires:

$$f(ci_i) = c_i, i = 1, 2$$
$$I_1 \in I_{c_1,imp} \wedge I_1 \in I_{c_2,exp}$$
$$\exists m_1 \in M_{f(ci_1),exp}, m_2 \in M_{f(ci_2),exp} : (m_1, f(ci_1), m_2) \in R_{impl}$$

Please note that $c_1 = c_2$ is allowed.

Coming with the component instances are method instances. The set of all these method instances is denoted by MI. There is a function $f1 : MI \rightarrow CI$ that identifies the component instance that a particular method instance belongs to. A second function $f2 : MI \rightarrow M$ identifies the method the instance belongs to. For $f1(mi_1) = ci_1$ and $f2(mi_1) = m_1$ it must hold:

$$f(ci_1) = c_1$$
$$m_1 \in M_{c_1,exp}$$

To be useful, R_{assemb} must follow some restrictions:

$$\forall I_j \in I_{suc} : \exists ci \in CI : I_j \in I_{f(ci),exp}$$
$$\forall ci \in CI : \forall I_j \in I_{f(ci),imp} : \exists ci_1 \in CI : (ci, ci_1, I_j) \in R_{assemb}$$

The first condition states that each interface needed for the overall system is provided by a component instance. The second condition requests that for each component instance ci selected there is also for each imported interface I_j a component instance ci_1 in the assembly that offers this interface, and ci calls methods of I_j of ci_1.

The *container* as the execution environment can be seen as another component instance, offering its services through well-defined interfaces. The only difference is that the container does not import any interface and there is just one instance of it. If we – without loss of generality – make the container the last component instance cin, this is expressed by $I_{f(cin),imp} = \{\}$ and for any $m_{k1} \in M_{f(cin),exp}$ and $m_{k2} \in M$, we have $(m_{k1}, f(cin), m_{k2}) \notin R_{impl}$.

In our simple example of `Math`, relation R_{assemb} selecting components for the imported interfaces simply states the fact that in this example there are no choices, but at least we have a component for each required interface:

$$R_{assemb} = \{(:SquareRoot, :BasicMath, Math)\}$$

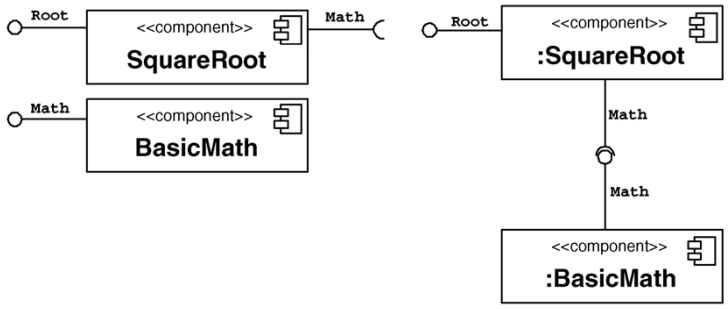

Fig. 3. Components `BasicMath` selected for imported interface `Math` of component `SquareRoot`

4.2 Assembly Matrix

As the goal is to reason about non-functional properties of the whole system under construction, the properties of all components must be considered. Taking again response time as a simple example, each component will have a response time for each of its methods (which depends on the available resources; see above). If one component calls another, the response time of the called method becomes part of the response time of the calling method. In total, the latter response time is accumulated from all calls to imported methods and the time used in the component itself.

As a consequence, it is necessary to additionally include the number of times a method is called into the model, not just the fact that it is called.

This is tricky, because it may (and often will) depend on the parameters of the call, sometimes also on the system state (in the form of return values of called methods). For a moment, we ignore this fact and simply assume an average number of calls.

Starting from the model of imported and exported interfaces introduced above, the relation R_{impl} must be extended so that it also tells how many calls of each imported method are caused by one call of a given exported method. This is similar to the lambda expressions for the sequenced operator calls in [5]. Omitting dependencies on internal state and parameter values and therefore assuming a constant number of calls for each method, this can be expressed by a matrix $A = (a_{ij}), 1 \leq i, j \leq |MI|, a_{ij} \in \mathbb{N}$ where each a_{ij} gives the number of calls of the imported method instance mi_j for a single call of exported method instance mi_i. It is understood that

$$a_{ij} = 0 \Leftrightarrow (\overbrace{f2(mi_i)}^{m_i}, \overbrace{f(f1(mi_i))}^{c_i}, \overbrace{f2(mi_j)}^{m_j}) \notin R_{impl}$$
$$a_{ij} > 0 \Leftrightarrow (\underbrace{f2(mi_i)}_{m_i}, \underbrace{f(f1(mi_i))}_{c_i}, \underbrace{f2(mi_j)}_{m_j}) \in R_{impl}$$

Assuming that there are no cycles, we can determine an ordering of all method instances $mi \in MI$ such that the matrix has the form shown below. Each method calls only methods coming later in the ordering.

$$A = \begin{pmatrix} 0 & & & & \\ 0 & 0 & & a_{ij} & \\ 0 & 0 & 0 & & \\ 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

In the first approach, no parameter dependency is represented. In reality, the entries of R_{impl} would more look like $(m_i, data_i, c, m_j, data_j)$. Even if the instances of *data* were again grouped into intervals and regions, this would significantly enlarge the matrix. Thus it is postponed to subsequent work.

For a given load (that is a vector $L = (l_j)$, where each l_j gives the number of end-user calls to method instance mi_j) (potentially per time unit, so giving a rate), we can easily calculate the total number of calls (TL for total load) of each method instance in the system:

$$TL = \sum_{i=1}^{|MI|} (A^i \cdot L) + L$$

Please note that this also includes the number of calls to the method instances of the container, which most likely will be found towards the end of the list of

mi_j. Having an average memory and CPU demand for each method, the total CPU consumption can already be estimated by vector multiplication of TL and a "CPU vector". However, this is only a very rough approximation, since the demand is only characterized by a single value, independent of parameters, and no intervals, distributions, or bounds are used. More work is required here.

This formal notion defines some kind of a "scaffolding" that we can now use to derive non-functional properties of the whole system under development.

4.3 Testing Non-functional Properties

In general, there will be more than one choice of R_{assemb}, because the same interface can be offered by different components. Furthermore, even with the same component, there can be several choices of instantiation (see figure 2). Given the matrix A (based on the number of calls as an additional information to R_{impl}), each choice can be evaluated. This can be done by the application assembler or by the runtime system. In the first case the assembler builds an application referring to fixed components, while in the second case the assembler only builds an application by referring to interfaces, leaving the selection of implementations to the runtime system. Of course, the available information is different in both cases, for example the runtime system has also the information of actual memory or load.

Let us assume that the user of the system requests an upper bound of t_{max} for the response time of one call to method instance mi_1. Using the formula introduced above we now just look at method instance mi_1 and use a load vector L_1 expressing one call to mi_1. We calculate the induced load TL_1 on the whole system. Knowing for each method instance mi_i its local calculation time [14], we derive an inequation that must be fulfilled for mi_1 if its response time $t_1 \leq t_{max}$.

$$L_1 = (1, 0, \ldots, 0)$$

$$TL_1 = \sum_{i=1}^{|MI|} (A^i \cdot L_1) + L_1$$

$$t_{max} \geq t_{1,local} + \sum_{j=2}^{|MI|} TL_{1,j} \cdot t_{j,local}$$

Now it is required that each component can name its resource requirements to achieve a given local processing time $t_{j,local}$. Here, we can refer to the formula that we introduced in Sect. 3.2.

[14] I.e. the time the method consumes without calls to other imported methods.

The formula above for t_{max} defines a search problem, as the values $t_{j,local}$ are not given, but are only constrained by the inequation. So, in a first step, values for the $t_{j,local}$ have to be chosen that fulfill the inequation [15] . The choice is based on the values recorded in table 1. The response times of a method cannot be arbitrary, but usually are from an interval. Even with an abundance of resources, they will not pass a lower bound. So, if the minimum values for all $t_{j,local}$ do not fulfill the inequation, this assembly cannot offer the requested response time. Otherwise, we must check whether the resources needed for the $t_{j,local}$ chosen are available. This information can be obtained by using the functions $f_{m_k}^{-1}$ that have been derived above in section 3.2.

$$f_{f2(mi_j)}^{-1}(t_{j,local}, \diamond) = \{(CPU_1, MEM_1), (CPU_2, MEM_2), \ldots\}$$

(The $\diamond$ stands for empty data, that is, the resources should be determined for any parameter value.) We choose one tuple according to any criteria, i.e. arbitrarily or with aim of minimizing the demands for one of the resources. Asking a resource manager (e.g. the operation system which is responsible for resource reservation), it can be decided whether the resource demands can be met or not. If they can be met, the NFP response time $\leq t_{max}$ can be achieved, and the resources required for that are known. If not, another tuple from the result of $f_{f2(mi_j)}^{-1}(t_{j,local}, \diamond)$ must be chosen. If the set of tuples has been exhausted, and the sum of the $t_{j,local}$ is not equal to t_{max}, at least one of the response times can still be increased, and the testing of the resources can be repeated. If all that fails, the assembly chosen exceeds the resources available or a given resource limit for the assembly. So a alternative assembly has to be constructed.

That means, different components supplying the imported interfaces have to be searched, yielding a new R_{assemb}, which has to be evaluated with the process described above. This process ends, when an assembly is found that meets the given constraints or all possible assemblies have been evaluated; then it has been ascertained, that for the given platform and with the given components the response time demanded can´t be achieved.

The formalism also allows to go the other way. That means to enquire the available resources and then use $f_{f2(mi_j)}$ to calculate the values of $t_{j,local}$ and finally the response time of method instance mi_1. However, this is considered to be less interesting, because the available resources are known only at runtime.

[15] Note, that this inequation is based on the assumption of a constant number of calls for each method (what can also be seen as an average number of calls derived from many different working conditions for a component); introducing parameter value dependency would complicate this equation severely.

5 Conclusion

A model has been developed that allows to describe components, their interfaces, methods, and the structure of a component assembly. Components offer interfaces and import interfaces, each of which is a set of methods. A component implementation decides which exported method calls which imported methods. The assembly consists of component instances that call each other. Their wiring is defined in terms of interfaces that one instance offers and another one imports. This information together with the frequency of imported-method calls is combined into a matrix. This matrix allows to test NFPs of an assembly, mapping them to resource requirements. If the resources are available, the NFPs can be achieved. If not, different assemblies must be constructed and again be tested.

Many extensions must be addressed in subsequent work: As mentioned several times, NFPs of methods depend on the parameters and the internal state, if it exists. Both of them can be of complex types, so their specification and modeling must be investigated in more detail. Whatever technique is used to derive the resource functions, complete knowledge can hardly be achieved (e.g. for a blackbox view, it would require exhaustive measurement). Instead, intervals for parameters should be identified where similar behavior can be observed. Then the resource functions can be defined in terms of these intervals. Matrix entries could be intervals, too.

Another goal is to achieve platform-independence of the resource specification. While this looks easy for memory, CPU needs a feasible unit like some kind instruction count or path length.

Obviously, NFPs other than response time must also be considered. At the moment, they must be investigated one by one, before a common treatment can be identified. Their nature is very different, so most likely a common treatment can only be achieved for classes of similar NFPs. However, we believe that the approach presented here is applicable to NFPs other than time and space as well, as long as they are additive. We will investigate this issue in further work.

Currently, we are developing a framework for determining the resource usage and selected NFPs of Enterprise Java Beans (EJB [12]) based on the open-source application server JBoss [7]. Additionally, a repository for storage and retrieval of component implementations and their descriptions – both functional and non-functional – is in a first development phase. In the end, this repository should be included in the design process, enabling the selection of components not only based on functions, but also based on NFPs.

Acknowledgement

The detailed comments of the three anonymous reviewers were extremely useful. Their help is greatly appreciated.

References

[1] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[2] Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. Towards pervasive treatment of non-functional properties at design and run-time. In *Submitted for publication at ICSSEA 2003*, 2003.

[3] Girish Brahnmath, Rajeev R. Raje, Andrew Olson, Barrett Bryant, Mikhail Auguston, and Carol Burt. A quality of service catalog for software components. In *Proc. Southeastern Software Engineering Conf. (Huntsville, Alabama, April)*, pages 513–520, 2002.

[4] Jean-Michel Bruel, editor. *Proc. 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*. Cépaduès-Éditions, June 2003.

[5] Merijn de Jonge, Johan Muskens, and Michel Chaudron. Scenario-based prediction of run-time resource consumption in component-based software systems. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, Portland, Oregon, May 2003.

[6] Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. The COMQUAD component model - enabling dynamic selection of implementations by weaving non-functional aspects. In *International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, 22 - 26 March 2004. ACM.

[7] JBOSS Group. Jboss application server website. URL: http://www.jboss.org.

[8] Frédéric Guidec and Nicolas Le Sommer. Towards resource consumption accounting and control in java: a practical experience. In Grzegorz Czajkowski and Jan Vitek, editors, *ECOOP 02 Workshop on Resource Management for Safe Languages (Malaga, Spain, June 10–14)*, 2002. URL http://www.ovmj.org/workshops/resman/.

[9] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Addison Wesley, 2003.

[10] Daniela Mania, John Murphy, and Jennifer McManis. Developing performance models from non-intrusive monitoring traces. In *Proc. of IT&T Annual Conf. (WIT, Ireland, Oct. 30-31)*, 2002.

[11] Florian Martin. PAG - an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[12] Sun Microsystems. Enterprise JavaBeans Specification, version 2.0. Final Release, 2001.

[13] Object Management Group OMG, http://www.omg.org/docs/formal/02-06-07.pdf. *CORBA 3.0 - IDL Syntax and Semantics chapter*, July 2002.

[14] Christoph Pohl and Steffen Göbel. Integrating orthogonal middleware functionality in components using interceptors. In *Proc. Kommunikation in Verteilten Systemen (KIVS'03, Leipzig)*, Leipzig, Germany, February 2003. To appear in Informatik Aktuell, Springer.

[15] RASC. Resource-Aware Software Components. URL http://www.univ-ubs.fr/valoria/Orcade/RASC.

[16] Simone Röttger and Steffen Zschaler. CQML$^+$: Enhancements to CQML. In Bruel [4], pages 43–56.

[17] Simone Röttger and Steffen Zschaler. Model-driven development for non-functional properties: Refinement through model transformation. TU Dresden, Fakultät Informatik, submitted for publication, August 2003.

[18] Murali Sitaraman. Compositional performance reasoning. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proc. 4th ICSE Workshop on Component-Based Software Engineering - Component Certification and System Prediction (Toronto, Canada, May 14-15)*. IEEE Computer Society, USA; Monash University, Australia; Carnegie Mellon University, USA, 2001. URL http://www.sei.cmu.edu/pacc/CBSE4-Proceedings.html.

[19] Murali Sitaraman, Greg Kulczycki, Joan Krone, William F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 3–10, 2001.

[20] Nicolas Le Sommer and Frédéric Guidec. A contract-based approach of resource-constrained software deployment. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD 2002)*, volume LNCS 2370, Berlin, Germany, June 2002. Springer.

[21] Sethu Sreerama, David Fleming, and Murali Sitaraman. Graceful object-based performance evolution. *Software - Practice and experience*, 27(1):111–122, January 1997.

[22] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley Publishing Company, second edition, 2002.

[23] Virginia Polytechnic Institute and State University. The resolve language. URL http://people.cs.vt.edu/~edwards/resolve/.

[24] Steffen Zschaler and Marcus Meyerhöfer. Explicit modelling of QoS-dependencies. In Bruel [4], pages 57–66.