

Tuning Parameters for Component Based Design with Memory Constraints

Y. Barbaix, K. Hermans, T. Holvoet, Y. Berbers

K.U.Leuven, Department of Computer Science, division DistriNet,
Celestijnenlaan 200A, 3001 Leuven, Belgium

E-mail: Yvan.Barbaix@cs.kuleuven.ac.be

Kris.Hermans@cs.kuleuven.ac.be

Tom.Holvoet@cs.kuleuven.ac.be

Yolande.Berbers@cs.kuleuven.ac.be

Introduction

Component Based Design (CBD) tries to bring software development to the level of real engineering. Software units (components) are used as building blocks for concrete systems. They hide component internals from the application developer (the component composer), which allows him to focus on building applications. This simplifies the integration process and improves development productivity by reusing existing and proven components.

Our experiences with the *embedded* community learn that this application domain too needs an increase in reuse of existing software and integrating it with as little effort as possible. However, in embedded systems, designers have to cope with resource constraints, such as memory, timing and bandwidth limitations. The reasons for these constraints are per unit production costs and high reliability and availability of the devices. Embedded systems are much harder to update (if at all) than common desktop applications running on general-purpose computers. Therefore, any good development process (in casu CBD) should deal with these constraints as early in system design as possible.

This paper focuses on memory constraints and gives our vision on how to integrate them in a component-based development process (called: CBDM, Component Based Design with Memory constraints). First a naïve approach to CBDM is presented. A case is introduced and serves as a vehicle to identify key problems. Next, we analyse memory consumption in a component system with its key influencing factors.

We propose to annotate a component at three levels in order to predict memory usage: at the method level, class level and component level¹. This prediction will be valid only in the context of interaction with other components and the given problem domain. The annotation allows the developer to tune and trim components in order to obtain a high degree of confidence that memory constraints will be met. This leads to a flexible, iterative process, which allows for what-if analysis during the component integration phase. Finally, discussion points are highlighted together with future research directions.

A naïve vision on CBDM

In general, a developer building applications based on components will use the following development pattern: first, he starts to analyse the problem and decomposes the problem domain into functionality and domain-specific subsystems; then, he searches component libraries or databases of available components to see if what he needs already exists; if it does, he takes the components, plugs them together and tries to deploy the new composition.

An example

Let us look at a simple example. Suppose a designer is given the task to build a new system. He analyses the problem and concludes that he could use a monitoring subsystem. He searches his component database, finds what he needs and composes it into the following subsystem (Figure 1).

¹ Note that, although we employ a Java syntax throughout this paper, we envisage a run-time environment that allows freeing memory by the programmer, and that has no garbage collection.

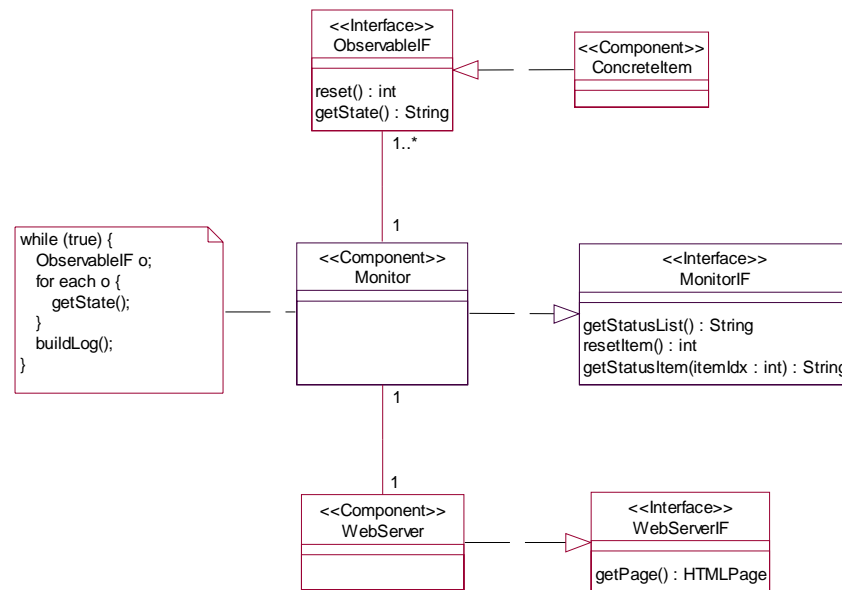


Figure 1: a remote diagnostics system

This is a simplified model of a system for remote diagnostics. Such a subsystem could be part of all kinds of embedded devices (routers, printers, UPS's), which can be monitored remotely via a standard web browser. The monitor component is an active entity: it continuously monitors its ConcreteItems and builds a log book. This log book can be fetched by a client via the WebServer component. ConcreteItem components are abstractions for anything that should be monitored. Depending on the application, ConcreteItems could be temperature, velocity, stock level, voltage, ...

Computing the memory requirements

During the integration phase, the developer will try to glue the components. But in order to verify if the system can run within the given memory constraints, he has to determine the system's memory usage. A naïve formula to use would be:

$$\left(| Total\ Memory\ Consumption | = \sum_{i=1}^n | memory\ consumption\ component_i | \right) < | available\ memory |$$

There are several reasons why this approach is very difficult at best:

- Is it possible to determine the memory consumption of an individual component and is this value a fixed number?
- Because of the dynamic behaviour of component interactions, taking the sum of memory consumption is not correct in order to determine the total memory needs.
- Is the total amount of memory a static number?
- How does the process evolve when this total does not match the given resource constraints?

For instance: it is impossible to say how much memory the MonitorComponent will consume as this is directly dependent on the amount of memory needed to produce a status string for a ConcreteItem. In conclusion, it is obvious to say that using this simple formula will not work in general.

Components annotated with parameter buttons

What we are proposing in the rest of this paper is twofold:

- to provide a means to describe the consumption of memory of one component;
- to make clear that a simple summation of component memory consumption is insufficient for our purposes, and instead an iterative approach with parameter tuning is put forward.

First we motivate the use of annotations for memory usage. Then, we propose to start memory annotations at the level of class methods, as this is the primitive unit of execution in an object environment. Finally, we overview the three levels of annotations: for methods, classes and components.

The level of annotation: abstraction vs. resource allocation

History of computer science demonstrates that various generations of programming languages add levels of abstractions in order to hide internal mechanisms of computers. These abstractions are introduced with the sole purpose of increasing the programmer's productivity. However, designing with constraints demands that we should be aware that resources are not unlimited. So a good understanding and incorporation of internals without breaking abstractions will be the solution. In other words, if we want to "manage" the memory a piece of software consumes, we need to understand how software uses memory and why and when software needs memory. This will force us to look at the low-levels of computer mechanisms.

Memory usage

We believe that understanding memory usage should start at the method level. Every method requires heap space for the code. Each time a method is invoked, there has to be an amount of stack space and possibly an extra amount of heap space for dynamic allocation of objects. The total amount of memory that is actually consumed varies from invocation to invocation, but the key issue here is that it is (to a limited extent) predictable and always deducible from the code.

Let's illustrate this idea with an example. Consider this method:

```
//do a large computation with a matrix of dimensions p and q
void largeComputation(Matrix m, int p, int q) {

    Row temp;
    ...
}
```

This method obviously needs space to store its code (probably on the heap), it requires stack space to push its parameters, and it requires additional heap space for new and temporal objects. Given the method's code and concrete values for p and q, it is possible to determine the actual amount of heap and stack space required. For example, suppose that the largeComputation needs a row to store temporal data. The amount of memory needed for this row is determined by parameter q (which is the amount of elements in the row).

In general, heap and stack space required for a method call is a function of its arguments (direct or indirect).

Annotations through "parameter buttons"

Estimating memory consumption is not a trivial problem. In the remainder of this paper, we introduce the term "**parameter buttons**", which identify those aspects of a method, class and component that directly influence memory consumption. The idea is that parameter buttons can be tuned to fit application requirements.

The use of parameter buttons hence leads to a new way of gluing components together while ensuring memory constraints of an application: *Component Based Development with Memory constraints*. It is an iterative process of tuning component parameter buttons and verifying constraints until the application requirements are met. In an ideal, future environment, this process should obviously be supported by an appropriate tool set.

1. Method level

Recall the previously defined method:

```
//do a large computation with a matrix of dimensions p and q
void largeComputation(Matrix m, int p, int q)
```

The memory usage of this method is influenced by several factors:

- Memory for the code (blueprint) will be allocated at the heap. This is a fixed amount and is compiler dependent.
- An actual call of this method requires stack space for pushing the invocation parameters, allocating local variables and storing results.
- Whenever this method needs to allocate new objects, it does this by reserving space on the heap (notice that in object-oriented technology, heap allocation is the domain of constructors).

The latter two sorts of memory usage are not standalone nor fixed. Instead, this may depend on a number of factors, such as the values of the invocation parameters, the algorithm that is used for the implementation, and so on.

Based on some first but non-trivial experiments, we believe that it is possible to capture these factors in a small set of so-called “method parameter buttons”. Examples of method parameter buttons are the method arguments (as in the example), but also local variables and temporary storage that is required by the employed algorithm.

For each parameter button, we define a “parameter button range” as the possible set of values of the parameter buttons.

The memory properties that are of importance can be defined as follows:

- \max_m : the maximum amount of memory needed to execute this method for the given parameter button ranges.
For instance, this is the maximum amount of stack space needed in case of recursion plus the maximum amount of heap needed to allocate new objects.
- Δ_m : this is the memory consumed (or freed) by this method and is also determined by the parameter button ranges (we call it memory flux). It can be defined as follows: $\Delta_m = (\text{memory usage at the end of the method}) - (\text{memory at the beginning of the method})$. $\Delta_m > 0$ means an allocation of memory, $\Delta_m < 0$ means a freeing of memory.

These two expressions are functions of (at least, cfr. infra) the method parameter buttons.

We can conclude that, given the fact that parameter button ranges are known (something the programmer determines during design), one can estimate the heap and stack space required to execute one invocation of this method with any valid arguments. During design, at a relatively early stage, playing with or tuning these parameter buttons ranges influences memory consumption and plays the major factor for consenting to application constraints.

Going back to the previous example, the developer can limit the number of rows and columns needed in the `largeComputation()` method to the values imposed by the problem to be solved. Based on this, it is possible to determine the memory constraints of the method in this particular case.

This early information is a big advantage with respect to traditional approaches where memory constraints are only checked against fully coded prototypes late in the development cycle.

How to deal with infinite loops ?

A lot of programs (if not all) contain infinite loops. Examples are: the “main loop”, server threads waiting for connecting clients, event dispatching threads waiting for user input events, ...

Consider the following example:

```
void main() {
    while (true) {
        c = readCommand();
        Case c of
            X: ... foo(...)...
            Y: ... bar(...)...
            ...
            Z: ...exit(0);
    }
}
```

When invoking `foo()` induces a memory flux $\Delta > 0$, then this implies that `main()` would need infinite memory space to guarantee proper working in all situations. Theoretically one could argue that \max_{main} and Δ_{main} strive to infinity, which makes it impossible to reason about fitting the memory constraints. In practice however, every function is developed with a well-defined use pattern in mind, derived from use cases, design documents, etc. So in real life these values will converge and one can determine these limits on the average, *given the proper use* of the function.

In other words, what we actually want is to limit the memory usage of the function `main()`. From a theoretical point of view this is infeasible. Therefore we must artificially introduce an upper bound on the memory consumption of `main()` by means of what we will call “proper use parameter buttons”. Note however that this step is potentially dangerous, because this upper bound is not necessarily enforced at run-time.

In conclusion, we extend our theory by stating that the memory quantities \max and Δ are determined by its “method parameter buttons” or its “proper use parameter buttons” (artificially implying an upper bound on memory usage). Once known or calculated for a given set of parameters, the memory constraints can be checked early in design.

Figure 2 gives a visual representation of this idea for the concrete example `largeComputation()`. The whole memory consumption is determined by the tuning parameters, in this case the maximum values for p and q (the dimensions of the matrix). By playing with the knobs (turning them around) during design, the developer can tweak the method until the matrix dimensions for the application are covered and until memory constraints are met.

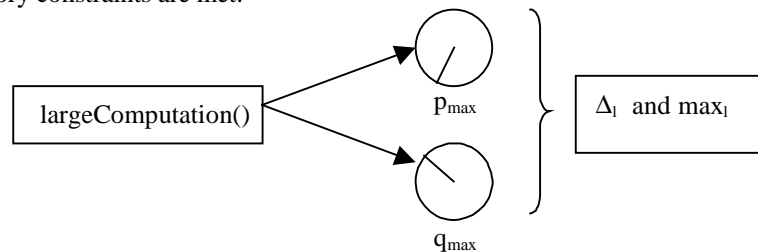


Figure 2. tuning parameter buttons for largeComputation

Figure 3 shows the general case.

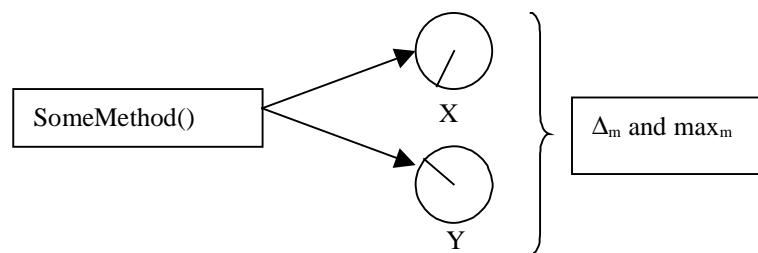


Figure 3. the memory usage of a method and its parameter buttons

`SomeMethod()` is determined by its parameters X and Y . This is generalised in the sense that these can be *method parameter buttons* or *proper use parameter buttons*. They are represented by turning knobs to stimulate the idea of tuning these parameter buttons in order to demonstrate the effect on memory consumption.

2. Class level

When several methods are grouped into a class, we see that many of the tuning parameter buttons are common to all methods. This obviously results from the fact that methods that are grouped into a class

share the object attributes they manipulate, or have some common arguments. Let's illustrate this concept with our example `largeComputation()`. As `largeComputation` does some computation with the data inside matrix `m`, other functions could exist that do some other calculations with this (kind of) matrix. For example, we could have a function `mainDiagonal()` that returns the main diagonal of the matrix:

```
void largeComputation(Matrix m, int p, int q);
int[] mainDiagonal(Matrix m, int p, int q);
```

As described in the previous section, the memory requirement of one invocation of `largeComputation()` is determined by the dimensions of the matrix `m` and the parameter buttons for this method are p_{max} and q_{max} . However, the amount of memory needed for one invocation of method `mainDiagonal()` will also depend on the dimensions of the matrix. As a result, the same parameter buttons for `largeComputation()` will tune the memory requirements of `mainDiagonal()`.

Besides the fact that multiple tightly coupled methods grouped together in a class can share parameter buttons, classes have something more. Let's take the classical stack example:

```
void init()
void push (Item e)
Item pop()
```

A class always defines (implicit or explicit) a use protocol for its methods. In the case of the stack this protocol states more or less that you have to initialise the stack before you can do anything and that you have to push first in order to be able to pop something.

We propose to extend such use protocols with respect to memory information by the introduction of protocol parameter buttons. These parameter buttons give an upper limit on the proper use of protocol of the class. For example, the protocol parameter button of the stack class could be: the maximum number of items on the stack is M . What makes protocol parameter buttons different from method parameter buttons is the fact that protocol parameter buttons work on the class level while method parameter buttons work on the instantiation (object) level.

In this section we have extended our concept of tuning one method for memory constraints to tuning a class for memory constraints. Figure 4 depicts this graphically. A set of parameter buttons (tuning, proper use, protocol) determines the memory consumption of a class and can be tuned early in design.

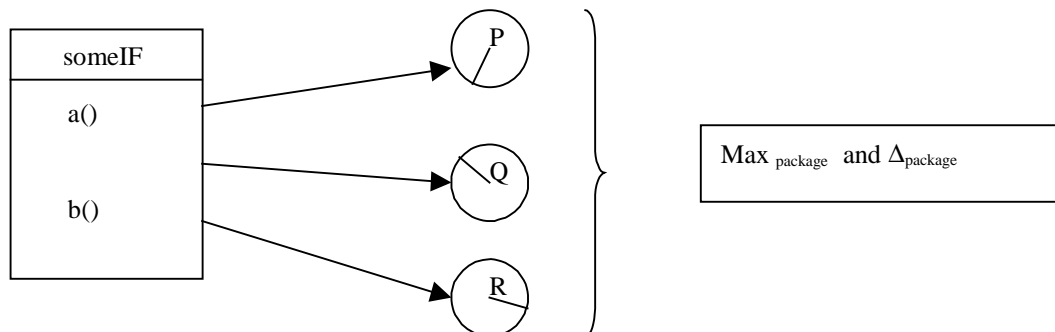


Figure 4. a class with its parameter buttons

3. Component level

A component is an entity that can contain multiple classes, much in the same way as methods can be grouped into classes. Therefore we can extend our theory in order to support components: we also introduce “protocol parameter button” on the component level. In what way the component level parameter buttons will have to be defined in the component interface is still an open issue -- especially as component development for embedded systems is still a research area. Figure 5 represents a component and their different parameter buttons graphically.

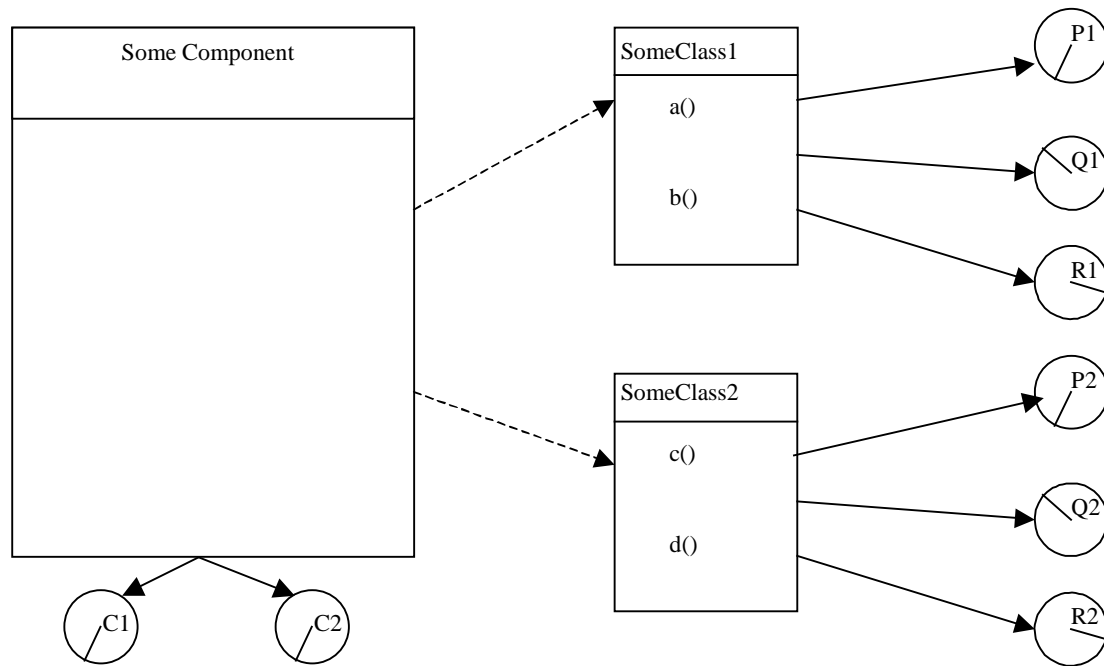


Figure 5. Component with parameter buttons

Open issues and known problems

- In our current work, we assumed a single threaded application. We still have to investigate in what way multiple threads will influence our method. Obviously, every thread will have its own stack which means that to a certain extend threads will not influence each other. However, as threads have a common memory space (heap) and threads communicate with each other, this will have some influence on memory. With the issue of multiple threads comes the problem of synchronisation. All this is work for future research.
- To enforce that the memory consumption will not exceed the requirements in cases where proper use parameter buttons are used, some form of run-time support for CBDM could be useful. Grzegorz Czajkowski and Thorsten von Eiken prove that such support can be implemented [1].
- The advantage of inheritance in program development is that we can postpone the actual implementation of a class and make abstractions of the concrete types of a class. As we need the target code to determine the memory requirements, inheritance opens new problems. On the other hand, the ultimate goal would be to use our approach in component development and it is not clear whether or not components should support inheritance
- Our approach assumes that the memory can be tweaked based on the parameters of the high-level programming language, but the exact amount of memory can only be defined once the target-code is available (or at least the exact translation of the source code to target code is known). In the popular case of Java environments with JIT compilers there is a problem. A JIT compiler uses standard target code (byte code) but makes a native version of the code at run-time. This makes it impossible to define the amount of memory needed for a piece of code during development. Notice that the whole idea of our approach is to enable the embedded systems developer to determine his memory requirements as early in the development process as possible.

Conclusion

This paper focused on the problem of predicting at design time the memory requirements of chunks of code. Our vision is one of a knob-based approach where software components can be tuned to the problem at hand and where memory requirements can be deduced much earlier than in traditional approaches. This CBDM approach will lead us to a flexible, iterative process, which allows for what-if analysis during integration phase.

Acknowledgements

This research is funded by the ITEA-DESS and the IWT-SEESCOA projects.

References

- [1] G. Czaikowski and T. von Eicken. *JRes: a Resource Accounting Interface for Java*. Proceedings of the conference on Object-Oriented Programming, Systems, languages and Applications, October 18-22, 1998, Vancouver, Canada
- [2] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998
- [3] Bruce Powel Douglass. *Real-Time UML, developing efficient objects for embedded systems*. Addison-Wesley, 1998
- [4] R. Hastings and B. Joyce. *Purify: Fast Detection of Memory Leaks and Access Errors*. Proceedings of the Winter USENIX Conference, January 1992