Component-Based Software Engineering for Resource-Constraint Systems:
What are the Needs?

D.K. Hammer and M.R.V. Chaudron
Dept. of Computing Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email: hammer@win.tue.nl and m.r.v.chaudron@tue.nl

Abstract

This position paper summarizes the most important problems that must be solved in order to establish a Component-Based Software Engineering (CBSE) discipline for resource-constraint systems. Resource-constraints are especially relevant for embedded systems (e.g. telecommunication systems and modern consumer products), real-time systems and dependable safety-critical systems, but can be encountered in virtually all types of systems.

In this paper, we define a number of research challenges in the form of requirements that are not yet met by contemporary component models. For each requirement, we explain its relevance and suggest directions for possible solutions.

We concentrate on the architectural level, since it is here that CBSE has the highest benefits in terms of adaptability and reuse. We also do not consider non-functional constraints in general, but only dependability constraints (timeliness, performance\(^1\), reliability, availability and security). The reason is that the latter directly affect the feasibility of a given component configuration, while this is less obvious for general constraints like scalability, maintainability and interoperability.

Requirement 1: Loose coupling.

Component models must support loose component coupling in order to achieve flexibility and configurability.

In present day component models like OMG Corba (Common Object Request Broker Architecture), Microsoft DCOM (Distributed Component Object Model) and Java RMI (Remote Method Invocation), peer-to-peer communication within a closely coupled network of components is the dominant form of interaction. The links with other components are either statically defined (i.e. directly programmed into the component in the regular object-oriented way) like for Corba or Java RMI, or dynamically obtained via a registry like for DCOM or the Corba Dynamic Invocation Interface (DII). In order to build systems that can be easily adapted, an interaction mechanism is needed that is less tightly coupled and thus more flexible. Figure 1 shows the four principal communication mechanisms that can be used to decouple the component interaction in space and time.

![Decoupling in Time and Space](image)

A sender and receiver are decoupled in time if the sender and receiver(s) do not have to agree on the timeliness of communication actions; i.e. send and receive actions may take place asynchronously. The space-dimension distinguishes communication mechanisms where the sender directs its messages to receiver(s) (hence the sender has a dependency on the identity or location of the receiver, and possibly also vice versa) versus communication where the sender and receiver do not know each others identity (anonymous communication).

All standard component models use direct coupling in the form of peer-to-peer networks. Only the DCOM and Corba DII models support more dynamic configurations and thus adaptability. Pipe and Filter architectures assume the buffering of data streams at the architectural level\(^2\) and decouple the components thus in time.

The Corba Event Notification Service and RMI also support the publisher-subscriber model (observer pattern)

---

\(^1\) Timeliness refers to constraints in form of deadlines, periods and offsets that are typical for hard real-time systems. Performance refers to average timing parameters like operations/transactions per unit of time that are usually encountered in soft real-time systems.

\(^2\) This buffer mechanism might be different from the buffering at the operating system level.
that gives more flexibility by allowing components to subscribe and unsubscribe to events at run-time. An implementation of a real-time publisher/subscriber middleware, called NDDS, is described in [8]. At the architectural level, this decouples the components in space, since the observer only receives events without knowing from where they come. In order to react to the event, it might, however, be necessary to query the observable for additional information.

Finally a blackboard model provides the weakest coupling and the most flexibility. As long as the data types are supported by the blackboard, components can freely connect and disconnect to the blackboard and put or get data at their own pace. The big advantages of such a component model are (1) the dynamic adaptability of the system and (2) the fostering of component reuse by the separation of computation (provided by the components) and coordination (provided by the shared data space of the blackboard). The challenge here is to define appropriate component models, to design Architectural Description Languages (ADL’s) that are based on coordination languages ([2] and [7]) and to extend these coordination languages in a way that allows also the specification and verification of dependability constraints. An interesting architecture that implements this component model in a distributed environment is e.g. the SPLICE system [1].

Another advantage of the publisher-subscriber and blackboard component model is the absence of implicit or explicit configuration information. Implicit or endogenous configuration information is built into the components and hinders their reuse and dynamic configuration. Modern ADL’s support the explicit or exogenous specification of configuration information outside the components. A typical example of such an ADL is Darwin [6].

**Requirement 2: End-to-end constraints.**

The component model must support the specification and verification of end-to-end dependability constraints. Dependability⁴ and especially time must become first-class design dimensions!

Most present-day ADL’s concentrate on functionality and do not support the specification and verification of dependability constraints. As a consequence, components that run perfect in isolation or in a particular configuration may fail in other configuration because of resource conflicts like memory overruns and race conditions. In other words, conventional components models do not support composability in resource-constraint contexts. The problem of composability with respect to timeliness is only solved for dependable hard real-time systems, where statically verifiable Time-Triggered Architectures (TTA) [5] can be used. This approach is, however, not usable for the majority of systems for the following reasons: (1) TTA systems are very expensive because of the worst-case assumption and usually only affordable for safety-critical systems; (2) TTA systems are very fragile because the assumption coverage must be 100% (i.e. the environment must be completely known at design time); and (3) the TTA approach is not suitable for soft-real time systems where the average performance is important and not the worst-case timing.

Real-time languages and especially object-oriented ones, have a tendency to specify timing constraints at the level of classes or objects. We claim that this is principally wrong for the following reasons:

- What counts are the end-to-end timing constraints⁵ that are defined in the requirements; all other timing constraints are artificial. If also the individual components must obey timing constraints, the real-time design space is considerably restricted and it might be impossible to find a feasible schedule.
- Components must be reusable in different contexts, i.e. on different execution platforms and with different timing constraints. Sometimes, the same component is even reused with different timing constraints within the same system.
- The obeying of timing constraints is a dynamic feature that depends on the actual invocation patterns of components which is usually only known at runtime. We also talk about emergent features that do not depend on individual components but on the interaction of a set of components and the available resources.

Similar arguments hold for all other dependability constraints. It is therefore meaningful to separate the definition of dependability and Quality of Service (QoS) constraints from the specification of the components. A convenient way to do this is to consider end-to-end transactions with non-functional constraints. In an object-oriented (UML) context, critical end-to-end transactions can be derived from the use cases, described by object interaction diagrams and refined together with the class hierarchy. A more detailed description of this approach is given in [4].

**Requirement 3: Resource requirements.**

The component specification must include resource requirements and interaction styles (protocols). Both should be specified in a platform-independent way.

In order to check whether a particular component configuration is composable in the sense that it not only

---

³ At the implementation level, the observable, of course, also holds references to its observers.
⁴ Dependability constraints are a special sort of non-functional constraints.
⁵ End-to-end constraints are defined between a stimulus from and a response to the environment.
implements the required functionality but also obeys all dependability constraints, we need to know the dynamic resource consumption of the component per service (interface operation). Resources include CPU, different types of memory (e.g. ROM, RAM and flash memory) and peripherals.

Figure 2 schematically depicts an approach for systematically specifying and analyzing resource requirements for component-based systems. A number of models (system, distribution, concurrency, dependability, etc.) are used to describe different aspects of the component system. It proposes that static resource requirements are specified per component, and dynamic properties per end-to-end transaction. Subsequently, the feasibility of the relevant dependability aspects should be verified by selecting and analyzing appropriate dependability models. This verification activity should start at the architectural level (based on estimates of the resource consumption) and continue via different levels of refinement down to the final implementation (based on measurements of the resource consumption).

**Figure 2: Verification of component composition.**

In order to make this approach workable, however, a number of non-trivial problems must be solved:

- The resource consumption of a component is difficult to determine since (1) it varies in time, (2) estimations are difficult and require a lot of experience, and (3) measurements need the component to be instrumented, which, in turn, influences its behavior. Memory and peripheral resources are relatively easy to estimate but CPU resources cause great difficulties.

- The CPU consumption of components is coupled via the various caches and pipelines of modern CISC (Complex Instruction Set Computers) processors. As a result, one has to rely either on very coarse estimates or on very detailed and elaborate simulation models. A lot of progress has been achieved in this area [9] but much more work is needed to come to approaches that are workable in practice.

- The strong dependency of the CPU consumption on the processor type and the components that execute on the same machine, make it difficult to specify the CPU usage in a platform independent way. The most general measure is cycles on a particular CPU because it is at least scalable by the clock frequency.

- The memory requirement of a component depends heavily on the programming language and the compiler. For binary components, these two factors are not relevant for the use of static memory. However, the efficient use of dynamic memory (stack and heap) still depends on the language constructs and the run-time structures generated by the compiler. With these restrictions, the most general measure is still the number of used bytes.

- For a good reliability model, not only the reliability figures of the hardware are important, but also the software reliability of the execution platform and the components themselves. Unfortunately, software reliability figures hardly exist because their estimation is very difficult and their evaluation is extremely time consuming.

- The construction of a particular dependability model is very time consuming, its accuracy is usually very limited and it is difficult to transpose it to other cases. In addition, the choice of the modeling approach and the construction of the actual model require a lot of highly specialized expertise.

In order to be composable, not only the static component interfaces must fit, but also the interaction styles (i.e. the sequence of service invocations over time) must fit. At the architectural level, all components must conform to the same architectural style, e.g. client/server or publisher/subscriber [3]. But also more complex behavioral constraints, usually described in form of protocols, might be relevant.

**Requirement 4: Dependable platform.**

The execution platform should support the implementation of dependability. The challenge is the combination of different dependability aspects!

Since the various dependability features are emergent, their implementation requires global mechanisms, i.e. mechanisms that affect all components. Typical examples are a global (distributed) scheduler to achieve timeliness, the replication of components to achieve reliability and the establishment of encrypting schemes and firewalls to achieve security. Another way to express this relation is by means of what was stated in the section about requirement 2: dependability requirements need to be defined over end-to-end transactions that invoke the services of many components. Since a chain is as strong as

---

6 Remember that testing can only reveal the presence of failures but never their absence.
its weakest link, the dependability of a transaction depends on the dependability of all services involved.

<table>
<thead>
<tr>
<th>Component</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$C_n$</td>
</tr>
</tbody>
</table>

Conceptual component interaction

Mutual dependencies

<table>
<thead>
<tr>
<th>Actual component interaction</th>
<th>Platform dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platform services</td>
</tr>
</tbody>
</table>

Execution Platform
Middleware & (Distributed) Operating System
Processor(s), Memory, Input/Output

Figure 3: Component environment.

It is sensible to factor out common features to the execution platform, i.e. to the middleware. The rationale is similar to that for the use of an operating system, i.e. a coherent, efficient and correct implementation. Especially if we are dealing with COTS components, it is virtually impossible that all components obey the same rules for implementing dependability. The platform should thus provide common services for scheduling, component replication, synchronization of component replicas, authentication, encryption and threat interception. A graphical representation of such a component environment is shown in figure 3.

The usability of such a component model depends, of course, on the standardization of the platform services. The specification of the common Corba services, like the naming service, the object trader service, the object transaction service the object security service and the concurrency service, is a first step into this direction. Also Microsoft offers a number of (non-standardized) services that build on DCOM. Examples are the directory service that implements a distributed registry, a messaging service that ensures message ordering, a transaction service that supports online transaction processing but not fault tolerance, and simple security services. Nevertheless, there are still many open issues and also the subsequent standardization of common platform services will need much time.

Requirement 5: Visual development environment.

Visual configuration and verification environments must support the use of component models. Any component model is as useful as the supporting tools. Except for simple cases, there is not yet much in this area. This can also be attributed to the immaturity of the area (most component models concentrate e.g. on functional properties) and the complexity of the issues involved.

Nevertheless, it is meaningful to sketch the ultimate goal of a visual component composition environment where fully specified COTS components can be selected (by various features) and composed. Based on the specification of the relevant end-to-end transactions, such an environment should calculate the functionality and non-functional properties of the overall system. In addition, it should select and verify the most critical traces belonging to these transactions as well as visualize the bottlenecks of the systems in order to support the designer or architect in redesigning the system, if necessary. It should also be possible to compose more complex components from simple ones, to derive their specifications automatically and to store them in the component repository.

Conclusions

We have tried to identify the most important open issues for CBSE of resource-constraint systems. This is, of course, only a coarse categorization and many more detailed problems need to be discussed. Nevertheless, we hope that this position paper helps in establishing a research agenda for this important software engineering area.

The main point will be the definition of reusable component models that take also resources and non-functional aspects into account.

Literature