Chapter 2

GUIDELINES FOR IDENTIFYING OBSTACLES WHEN COMPOSING DISTRIBUTED SYSTEMS FROM COMPONENTS

Mehmet Akşit and Lodewijk Bergmans
TRESE Group, Department of Computer Science, University of Twente, postbox 217, 7500 AE, Enschede, The Netherlands. email: {aksit, bergmans}@cs.utwente.nl, www: http://trese.cs.utwente.nl

Keywords: Distributed systems, aspects of computer science sub-domains, obstacles in component composition, research in component composition

Abstract: Component-oriented programming enables software engineers to implement complex applications from a set of pre-defined components. Although this technique has several advantages, experiences show that effective composition of components remains a difficult task. This is especially true if the software system is physically distributed. This chapter provides a set of guidelines to identify the obstacles that software engineers may experience while designing distributed systems using the current component technology. To this aim, the computer science domain is divided into several sub-domains, and each sub-domain is described using its important aspects. Further, each aspect is analyzed with respect to the current component technology. This analysis helps software engineers to identify the possible obstacles for each aspect of a sub-domain. The chapter concludes with references to the relevant research activities that are presented in this book.
1. Definitions

1.1 Middleware Systems

To identify the obstacles related to composing components, we will refer to the distributed system architecture shown in Figure 1. From an abstract view, a distributed system can be divided into two layers: software applications and the middleware. We assume that software applications provide services to an environment, which is considered external to the software system. Application services can be considered specific, evolving and diverse. The middleware abstracts the underlying computing and networking technology and provides services that are required by most of applications.

![Diagram](image)

*Figure 1: Reference middleware architecture*

The main motivation in adopting a middleware is economics. Instead of repeatedly implementing services that are required by most distributed applications each time an application is developed, it is more economic to provide these generic services by the middleware system. For example, almost every distributed application requires a name server and remote invocation mechanism. Many distributed applications require transaction and security services. To determine the services required for a particular middleware system, it is necessary to enumerate the services that are
required by most applications likely to be installed on the middleware. Accordingly, the middleware system should provide these services.

1.2 The Obstacles Caused by Complexity and Evolution of Applications

Unfortunately, complexity and evolution of applications make designing distributed applications a difficult task.

Complexity may hinder a proper decomposition of the software systems into autonomous modules. Moreover, some aspects of the applications may not be expressed sufficiently by the adopted design and/or language models. We term the first problem as the decomposition problem and the latter as the lack of expression power problem.

We observe the following three affects of evolution: Firstly, the application domain of the middleware technology grows steadily. For instance, nomadic and agent-based computing are some examples of new developments in the area of distributed systems. These new applications generally require extensions to the current middleware services.

Secondly, demands for extensions to existing services require modifications to middleware. For example, most business applications today require support for flexible transactions, whereas current middleware systems generally provide strict serialization and recovery.

Finally, it is becoming more and more common that middleware systems offer services of different quality. These, the so-called quality of services can be defined in terms of various parameters such as performance, reliability, and security. The users of a system can be allowed to select the required quality of a service with respect to a certain cost.

The middleware designers may attempt to solve the above-mentioned problems by implementing a dedicated service for each particular service demand. Since demands are evolving and diverse, this would require continuous implementations of many services, which is unfeasible. Instead of implementing a dedicated set of application services, it may be more feasible to compose application services from simpler components that correspond to the fundamental aspects of the application being designed. Obviously component composition here plays a major role. The difficulties that the designers may experience in composing components are termed as the composition problem.
1.3 Identification of Obstacles Using Domain Analysis

In this chapter we adopt domain analysis techniques to identify the obstacles in designing distributed systems\(^1\). Figure 2 illustrates the domain analysis process adopted in this chapter. The first stage is to formulate the technical problems in the requirement specification. Second, these problems are used to select the corresponding solution domains. Third, if the solution domains are found, then the aspects of these domains are determined. Fourth, the expected obstacles are identified for each aspect. Finally, the obstacles are brought into the context of the application.

![Diagram](image)

*Figure 2: The domain analysis process*

In general, the domain of a software system can be intuitively divided into three, possibly overlapping categories:

- Application domain;
- Mathematical domain;
- Computer science domain.

The application domain corresponds to the specific concepts of the application being developed. Assume for example that we want to design a distributed container transport system, which has two main tasks: allocation of containers to the vessels, and creating route plans for the vessels to transport containers among seaports. The application domain here deals with the specific application features of a container transport system such as modeling containers, vessels and seaports.

The mathematical domain deals with the concepts studied within the area of mathematical sciences. In the container transport system example, allocation of containers to vessels and routing vessels among seaports can be seen as mathematical optimization problems.

Since we are interested in software realization techniques, the topics studied in computer science must be considered as well. In the container

\(^1\) For a more detailed description of domains, the reader may refer to chapter 1 of this book.
transport system example, a possible computer science domain is distributed systems since the container transport system is expected to run on multiple sites, such as at vessels and seaports.

Application domains are generally very diverse and therefore it is not possible to reason about the domain of an application without precisely specifying it. The mathematical domain is considered out of the scope of this chapter. In the following sections, the main focus will be on the computer science domain.

The remaining sections of this chapter are organized as follows. The assumptions in analyzing the computer science domain are presented in the following section. Further, aspects general to all sub-domains are introduced and the expected obstacles per aspect are identified. The obstacles are printed as underlined and in Italics. In section 3, the computer science domain is divided into several sub-domains. Each sub-domain is described by using its important aspects and for each aspect the expected obstacles are described. Finally, section 4 gives conclusions. Appendix summarizes the identified obstacles and refers to the relevant sections of this chapter.

2. IDENTIFYING THE OBSTACLES BY ANALYZING THE COMPUTER SCIENCE DOMAIN

2.1 Assumptions

To identify the obstacles of using components from the computer science perspective, we make the following assumptions:

- The computer science domain can be decomposed into sub-domains such as application generators, concurrent processing, constraint systems, control systems, distributed systems and real-time systems.
- The level of detail of each sub-domain can be quite different. For example, the domain of concurrency and synchronization can be considered to be more basic than application generator design.
- Each sub-domain is specified in terms of its aspects. The aspects of a domain are the important features that distinguish that domain from others.
- The intention is to provide intuitive and somewhat historical classification of the computer science domain. To further abstract the aspects of sub-domains may result in too abstract models (such as
lambda calculus), which we consider less suitable for the purpose of this chapter.

- While we are listing the aspects of a sub-domain, we prefer to include aspects only specific to that sub-domain, although in practice, very few pure domain-specific applications are developed. For example, although almost all distributed systems provide some degree of concurrency, we will treat concurrency separate from the distribution aspects.

- While defining the computer science sub-domains, we do not intend to analyze application specific issues but rather we want to know what kind of computation models are needed to support applications in that sub-domain. For example, when we talk about the security aspects in distributed systems and try to identify the problems related to these, we do not intend to assess the basic security enforcement techniques such as encryption, decryption and digital signatures. But rather, we would like to assess the applicability of the component model to support secure distributed systems.

- Some aspects are applicable to all sub-domains.

2.2 The Common Aspects of all Domains

The aspects described in subsections 2.2.1 to 2.2.7 are general and maybe used in supporting all kinds of distributed applications and middleware systems.

2.2.1 Components, Objects and Classes

Components are defined as autonomous software modules with well-defined interfaces. In the literature the term component generally refers to the abstractions provided by the enabling technology such as CORBA, DCOM, OLE, ActiveX and JavaBeans [17]. Objects are instantiations of classes, provide encapsulation and are characterized by well-defined interfaces. Component and objects are related to each other in that any composite and autonomous object structure, in principle, can be considered as a component. The component concept is general and suitable for constructing distributed applications and middleware systems.

---

2 We will use the term component for components provided by the enabling technology, and for autonomous objects specified by an object-oriented language.
2.2.2 Interface Declarations and Type Checking

One of the fundamental characteristics of components is the specification of the functional interface. For example, CORBA introduces the Interface Definition Language (IDL) for specifying the interfaces of CORBA components. Strongly typed object-oriented languages provide type-checking mechanism based on the object (class) interface concept. Further, class hierarchies can be used to define sub-type hierarchies.

Although typing the interface of components is useful for early detecting the interaction errors, problems can be experienced when all the meaningful combinations of components have to be declared as a separate type module. In this case, the designers may be forced to declare a large number of type modules.

Assume for example that a car game is to be constructed from components. The Abstract Factory pattern [10] is used for creating various car models in a flexible way. In this pattern, the factory component provides a set of operations to create consistent car components so that a car model can be safely composed from these components. Here, the factory component can be considered as the type of a consistent car model. As a result, for each version of a car model a different factory is necessary. Since a car model may have many different versions, it is necessary to define many factories. This problem is termed as the excessive type declarations problem.

2.2.3 Encapsulation and Multiple Interfaces

Encapsulation is an essential property of components and supported by all component models. Depending on the context, distributed applications may require some means to control the visibility of operations of a component.

Assume for example that a mail component migrates over a network passing through different layers. This generally requires changing the interface of the mail component based on its context. For example, the mail sender must have an access to the interface for creating the mail attributes such as the content. The mail-system layer must have an access to the interface of the mail component for approving and delivering the mail. It is not desirable, for example that the mail sender approves the mail and the mail system reads the mail content. This means that every mail component must be able to check the identity of the sender of a request, for example, before returning the mail content. The current component models, however, may have difficulties in detecting the sender of a request. Moreover, if the view checking is realized in the implementation of an operation, then the view checking and operation semantics become not separable. This makes a
separate extension of the view checking mechanism and/or operation semantics difficult. Note that introducing a different implementation component for every interface cannot provide the necessary solution. This is because there must be one component, with a single identity, and a single state, which behaves differently according to the way it is being interfaced. This problem is termed as the \textit{multiple-views} problem [2][5][7].

2.2.4 Message Passing

Message passing is the basic mechanism to express coordination among components. Applications may demand flexible message passing semantics from the middleware. Assume for example that an office workflow system has to be designed for supporting various office activities such as reporting, planning, internal request handling, agenda management and electronic meeting. Several of these activities must be executed concurrently. Further, electronic meetings may require stream-based communication.

In current component models, reuse and extensibility issues focus on extending and redefining the features of components. However, the concept of message passing, although a key feature of components, is not subject to such interest. Instead, the semantics of message passing are fixed, or can, at best, be picked from a small set of fixed semantics. Examples are the remote procedure call mechanism, asynchronous message passing, broadcast and multicast, and atomic message send semantics.

These fixed semantics cannot be tailored to model application-specific interaction mechanisms. To model alternative interaction mechanisms, code for implementing these must be added to all implementations of components participating in the interaction, provided it is possible. It is even harder to abstract interaction mechanisms and reuse this abstraction. Concluding, a mechanism, which can implement tailored semantics for message passing is required to solve these problems. Such a solution should allow the application of component-oriented techniques to obtain extensibility and reusability. This is termed as the \textit{fixed message passing semantics} problem.

2.2.5 Inheritance and Aggregations

Generally, behavioral composition of components is realized using inheritance and/or aggregation mechanisms.

Inheritance provides a compile-time \textit{XOR-signature} composition of operations of components in a transitive way. An \textit{XOR-signature} composition means that any inherited or self-defined operation of a component may be invoked exclusively and independently. A \textit{transitive-composition} means that the inherited operations are automatically available
to the subclasses of the component. The pseudo variable \textit{self} \(^3\) is essential, and used to refer to the component which received the operation call.

In case of an aggregation, the composed component should aggregate the components that are needed to be composed. This is for example exemplified by various design patterns such as Bridge, State and Strategy [10]. A programming effort is necessary to implement \textit{XOR-signature} composition such as declaring the necessary operations at the interface of the aggregating component and forwarding the calls on these operations to the aggregated components.

Inheritance and aggregation show different run-time and visibility characteristics. Both mechanisms have advantages and disadvantages. The main advantage of inheritance is that it provides a transitive reuse mechanism. Further inheritance is generally more efficient to implement than component aggregation. Disadvantages of inheritance are that inheritance cannot cope with run-time extensions and requires knowledge about the implementation of components. Further, in case of multiple inheritance name conflicts of the inherited operations may occur.

One important advantage of aggregation is that the aggregating component depends only on the interface operations of the aggregated components. This simplifies the design and improves the ease of reuse. Further, the aggregated components can be easily changed at run-time. Because of these advantages, most component models prefer aggregations to inheritance.

The main disadvantage aggregation is the lack of support for a transitive reuse mechanism. For example, sometimes in a distributed system the interface of a component cannot be fixed. Implementation improvements or migrating to a different platform may require extensions to the interface of a component. The Bridge and Strategy patterns [15] can be used to define components with dynamically changing implementations\(^4\). These patterns represent the alternative implementations as aggregated components, which can be changed at run-time. There are, however, a number of problems with this approach [3]. First, the aggregating component must declare all the operations explicitly and forward the calls to the aggregated components. This is an error-prone task. Inheritance allows all the inherited operations available transitively without necessarily declaring them. However, inheritance provides only a compile-time extension. Second, declaring operations at the interface of the aggregating component results in fixing the

\(^3\) The pseudo variable \textit{self} is called \textit{this} in C++ and Java, and \textit{current} in Eiffel.

\(^4\) The Command pattern [10] can be considered more suitable in implementing components with changing interfaces. This pattern provides a limited reflection on messages. Reflection is discussed in subsection 2.2.7.
number of operations that a component can provide. Therefore, the
operations of a component cannot be easily extended at run-time. Finally,
the aggregated components cannot access the aggregating component
directly by invoking on self. Although the pseudo variable self can be
simulated in the implementation of a component, the programmers may have
to deal with two inconsistent pseudo variables: one defined for the
components and the other defined by the implementation language, such as
Java.

2.2.6 Delegation

Delegation is introduced as an alternative to inheritance [11]. In
delegation, if a component cannot respond to a particular request of its
client, then it delegates this request to one or more designated components.
One of the designated components may execute the request on behalf of the
delegating component. Further, the designated component can refer to the
delegating component by calling on the pseudo variable self. Delegation is
similar to inheritance; the designated component behaves like the super-class
of the delegating component. Delegation can extend the interface of a
component if the component delegates the requests to another component,
which extends the interface. Delegation eliminates the need of declaring the
interface operations explicitly. Further, compared to aggregation, delegation
provides the pseudo-variable self so that the designated components can
refer to their delegating component by invoking on self.

An additional advantage of delegation is to share a common behavior at
the component level; the shared component may have a have state, which can
affect the shared behavior. Assume for example that in a distributed system
multiple components provide the operation checkSecurity. All these
components share the same implementation of checkSecurity. The
implementation of checkSecurity refers to an access-list to verify
invocations. It is desired that this access-list be encapsulated by the
operation checkSecurity. Delegation can easily implement such a
requirement. The shared designated object should then implement the
operation checkSecurity and encapsulate the access-list. It is not easy to
implement such a mechanism using inheritance because the operation
checkSecurity must then be inherited from the super-class and the super-

5 Inheritance suffers from the same problem. In strongly typed languages, generally a
compile-time error is generated if, say component C1 is replaced by an instance of its sub-
class, say component C2, and if an extended operation defined in C2 is invoked on C1. To
eliminate these errors, generally typecasting is used.

6 This is only possible if the adopted language does not generate typing errors, when the
new operation is called.
class must also encapsulate the access-list. Classes are not optimized for storing and encapsulating state. This is termed as the sharing behavior with state problem [2]. Unfortunately only a few languages support delegation [18].

One disadvantage of current implementations of delegation is that they cannot enable or disable the delegation process, for example, based on a condition of the delegating component. This may be necessary if the implementation of a component has to be adapted based on certain conditions. For example, in distributed systems a conditional delegation mechanism could be useful in adapting the protocols based on the quality of service requirements. Although the Bridge and Strategy patterns provide a similar functionality, they are aggregation-based and therefore do not provide the advantages of delegation as discussed in this subsection. In a conditional delegation mechanism, the designated components could implement the alternative protocols, and the condition of the delegating component would enable the designated component which implements the most suitable protocol. This is called the lack of support for dynamic composition problem.

2.2.7 System Interface and Layering

The interface of middleware systems must be fixed to achieve compatibility and portability. As stated in the previous sections, however, the application domain of the middleware technology grows steadily. Generally, this requires extensions to the middleware services. We term this as the unmatched system functions problem.

The application programmer may deal with this problem in three ways:

- Implementing the service at the application level. This approach causes a repetitive re-implementation of the common service for every similar application.

- Implementing the service as an application service. Existing middleware systems provide certain application services that can be explicitly called by applications if needed. These application services are defined at the same level as applications. This approach, is therefore, not desirable if the service must be transparent to the application program.

- Implementing the service within the middleware. This approach is difficult to realize because generally middleware systems are closed systems and do not provide facilities to extend them.

One of the reasons why middleware systems are closed systems is due to transparency of certain services. Middleware systems are organized in layers
and layers provide transparency in that certain aspects of a layer are invisible to the "higher-level" layers. Typical examples are location transparency, replication transparency, failure transparency, etc.

Although transparency of certain services eases application programming, it works against extensibility for two reasons: First, to be able to implement new transparent services, it must be possible to introduce new transparent layers into the system without re-defining the existing ones. For example, certain systems may require a dedicated transparent security layer, which might not be foreseen before. In case of a closed middleware system, however, this is not possible. Second, to be able to cope with the evolving application demands, it must be possible to adapt the services of a layer in a controlled manner. Transparency of certain aspects, however, may hinder the adaptation of certain services.

In the literature reflective systems are proposed to "weaken transparency" in a controlled manner [19]. A reflective system is a system which incorporates models representing (aspects of) itself. This self-representation is casually connected to the reflected entity, and therefore, makes it possible for the system to answer questions about itself and supports actions on itself. Reflective computation is the behavior exhibited by a reflective system.

The term reflection was introduced by [16] as a technique to structure and organize self-modifying procedures and functions. In [12] reflection was applied within the context of object-orientation. A considerable amount of work has been done in reflection techniques, for example, in concurrent computation, distributed system structuring and middleware, programming language design, real-time systems, and in network design [20][6]. Conventional component models do not provide adequate support for reflective system development. This is termed as the lack of support for reflection problem.

Recently, several attempts have been made to provide message reflection within a middleware. The so-called portable interceptors can be inserted into a middleware, which enables the programmers to access, test and modify the messages. Reflective computation is an active research area and there are a number of open questions:

- Which aspects of components must be reflected? For example the portable interceptors in CORBA are only defined for a limited set of interactions within the middleware. It is currently not clear how many interceptors are necessary at which levels. Moreover, intercepting provides only a limited degree of reflection.

- What are the suitable abstraction techniques? How can reflection be managed? For example the main abstraction of portable interceptors is
the representation of messages. This provides only a limited reflective capability.

- What are the suitable techniques to specify and enforce causal connections among the reflective levels? For example in the portable interceptor approach, this is basically defined at the level of manipulating messages.

- What is the affect of evolution to the reflective part of components? Can reflective parts easily evolve as well?

- What kinds of compositional mechanisms must be provided at the reflective layer? This is an important issue if multiple dependent properties are reflected. Since these properties are related to each other, suitable compositional mechanisms must be provided at the reflective level as well.

- If a certain property is shared among multiple components and/or levels, how can this property be reflected?

Especially the last three items pose problems and can not be solved adequately using the current reflection techniques. For example, in the portable-interceptors approach, it is not clear how the processing of reflected messages at the reflective-level can be composed. Consider for example the quality of service properties of a typical middleware system. These properties generally refer to multiple components in multiple layers and therefore cannot be easily captured by reflecting a single message, component or a layer.

3. THE COMPUTER SCIENCE SUB-DOMAINS

Based on the assumptions given in the previous section, we have selected the following sub-domains: application generators, concurrency and synchronization, constraint Systems, control systems, distributed processing and real-time systems. Obviously, practical software systems may include several of these sub-domains.

3.1 Application Generators

Figure 3 depicts a typical application generator (AG) architecture. AGs are used to construct software in a particular domain. AGs usually adopt a high level specification language in which the application can be described. The application domain needs to be well understood to allow applications to be described at a higher level of abstraction. AGs incorporate some default information in their application domains and thereby let the programmers
concentrate on the specific aspects of their programs. Based on the user’s specifications and default knowledge, AGs construct the application in an executable (general purpose) language.

![Diagram of application generators](image)

*Figure 3: A block diagram of application generators*

In middleware systems, AGs are used to generate stubs; stubs are software modules that link application-level functions to the functions of the middleware layer in a transparent manner.

In principle, AGs can be used in distributed systems as a general mechanism to hide certain implementation aspects of applications. For example, the application programmer may express his/her specific security needs in a *directive file*. A suitable AG can interpret this specification and bind the application to some dedicated services in a transparent way. The subsections 3.1.1 and 3.1.2 describe the important aspects of AGs.

### 3.1.1 Problem Domain Specification

The specification language of an AG must be expressive enough to describe all the important features in its domain.

Problem domain specification and implementation by itself is a complete analysis and design problem. Therefore, depending on the domain, the designer of an AG may suffer one or more of the problems listed in this chapter.

Secondly, while designing and reusing AGs, designers may experience the *arbitrary composition* problem, which is explained in the following:

Assume for example that we would like to generate the interaction protocol of a component from a specification language. If the protocol has various different versions and/or the future extensions are likely to occur, we may want to have composition mechanisms for this specification language. Reusing and composing these specifications using the standard class inheritance or component composition mechanisms may not be appropriate.

---

7 This is similar to the *arbitrary inheritance* problem which was described in [2][3].
since specifications may include specific constructs which cannot be supported by the standard inheritance and composition mechanisms.

3.1.2 Code Generation

The code generator requires that all the necessary details be provided in the input specifications. The generated code must be complete enough to execute on current middleware platforms in an efficient way. This requires realizable and precise models.

On the other hand, it is not desired to overload the programmer with too many details. Assume, for example that we like to optimize the remote accesses by considering the application characteristics, such as access rates, size of data, as much as possible. If the user does not give any details, it may be difficult for the AG to generate an efficient code. In general, the performance problem of the generated application cannot be improved only by the compiler of the generated code. There is a need for problem-domain specific optimization techniques, which have to be provided by the code generator of the AG.

3.2 Concurrency and Synchronization

Concurrent processing can be defined as a parallel or time-multiplexed execution of one or more operations, which can be, but are not necessarily, data-dependent. Whenever the execution of one operation is started before the execution of another operation is completed, the two executions are concurrent. The basic aspects in designing concurrent systems are presented in subsections 3.2.1 and 3.2.2.

3.2.1 Creating Concurrency

This can be done either explicitly through special constructs (e.g. fork [8]), or implicitly, as the result of certain message passing semantics. For example, when applying asynchronous message passing, the thread that initiates the call proceeds concurrently with the thread executing the call. The creation of new active objects is an additional possibility for creating concurrency.

Although some methods and programming languages provide a variety of message sending constructs, in general the semantics of message sending are fixed and cannot be tailored to particular needs. In section 2.2.4, this was defined as fixed message passing semantics problem.
3.2.2 Synchronization of Concurrent Executions

The activities of concurrently executing processes can be totally independent, but they may also interact at certain times. This interaction may involve the exchange of data, such as producer-consumer interaction, or can be pure synchronization between two or more processes, often with the aim of safely sharing resources.

Three approaches for integrating concurrency and component models can be distinguished:

- **Orthogonal approach**: Components and threads are completely independent, creation of threads and synchronization are achieved through special statements, such as the conventional forks and semaphores.

- **Homogenous approach**: Every component is considered to be active, and takes care of its own synchronization; components are the unit of concurrency.

- **Heterogeneous approach**: Adopts both passive and active components. Passive components do not perform any synchronization on incoming requests.

The homogeneous approach is well integrated with the component model. However, the combination of component composition with synchronization introduces new problems. This is termed as the composition versus synchronization problem.

Assume for example that the message invocation performance of a middleware system has to be improved by dynamically switching between various protocols. For instance, if a client component invokes on its server only once, then relatively, the TCP/IP protocol is considered as the most efficient implementation. However, if the same client component starts to invoke on multiple servers repeatedly, then the multicast protocol is considered the more efficient. Studies show that [15] in current middleware systems, however, switching from one protocol to the other at run-time is generally not possible. This is because, the synchronization action is distributed to various components and layers. As a result, new protocols cannot be easily incorporated at run-time without replacing a considerable part of the middleware system.

---

8 This is similar to the inheritance anomaly problem [13]. An extensive discussion of these anomalies is given in [4].
3.3 Constraint Systems

In constraint systems applications are modeled using components and relations between these components that need to be enforced. These relations, in general, are constraints on the attribute values of the components and expressed in formulas. As constraints are often related to each other, changing the value of one attribute can have a considerable effect on the attributes of other components.

Assume for example that a distributed geographic information system has to be designed. This system allows creation, modification and deletion of geographic data by multiple users. Geographic data is constraint by the topology. In addition, the physical characteristics the artifacts, such as houses and roads have to be considered. The system has to maintain the constraints when certain artifacts are created or modified.

Subsections 3.3.1 to 3.3.3 present the important aspects of designing constraint systems.

3.3.1 Constraint Enforcement Strategies

The constraint enforcement system must determine which component's attribute needs to be changed when a constraint is violated. Assume that the constraint is expressed as $x + y = z$. Here, if $z$ is the independent variable, then the constraint can be enforced by updating the dependant variables $x$ and $y$. If, however, none of the variables are independent, then the problem becomes even more complicated, since the constraint system must adopt a more complicated constraint enforcement strategy.

Since constraint enforcement among components can be defined as a coordination activity, components must represent the coordinated behavior explicitly. The Observer and Mediator patterns [10] for example, can be used to detect a change and enforce the constraints, respectively. The Mediator pattern encapsulates the implementation of the constraint enforcement strategy.

The designers can implement the constraint enforcement strategy at the application, application service or middleware levels. Here, the issues to be concerned are similar to the ones that were discussed in 2.2.7.

An important question here is how to implement the constraint enforcement strategy in a reusable and extensible manner. Generally, constraint enforcement is implemented by sending the necessary messages to the participating components. For example, the interaction pattern can be implemented by using the Mediator pattern. The interaction is mainly based on message send semantics, where the mediator component transmits messages one by one to the so-called colleague components. We consider
the message send model as being too low-level, because it can only specify communications that involve two partner components at a time and its semantics cannot be easily extended. Mechanisms like inheritance, aggregation and delegation only support the construction and behavior of components but not the abstraction of communication among components. These mechanisms therefore fail in abstracting patterns of messages and larger scale synchronization involving more than just a pair of components. For example, it is not trivial to extend the mediator component for the purpose of extending the constraint enforcement strategy. This is termed as the lack of support for coordinated behavior problem.

3.3.2 Conflicting Constraints

Just as each constraint can be related to multiple components, each component can be related to multiple constraints. This can result in conflicting constraints when the value ranges resulting from the constraints are not overlapping. One solution is to assign priorities to constraints and let the constraint with the highest priority be enforced.

If the constraint system is implemented as an application service or provided transparently by the middleware system, conflict resolution strategies may not be replaced or extended easily. This was termed in section 2.2.7 as the unmatched system functions problem.

3.3.3 Reuse of Constraints

The current component models do not support the use of constraints as an explicit feature. If constraints are added to components as attributes, then it may be difficult to reuse and extend them by using the current composition mechanisms. In section 3.1.1, this was termed the arbitrary composition problem.

Note that if the constraint system has to be generated automatically, then it is a special kind of application generator and includes the aspects of application generator design.

3.4 Control Systems

As shown by Figure 4, a (feedback) control system includes sensors and actuators to influence (or control) the behavior of the controlled system. Each measured value of the controlled system is compared with its associated desired value and based on the discrepancies the parameters for the related actuators are calculated and engaged. The controlled system may be influenced by several other sources besides the control system. In
addition to the controlled system, modeling these sources and reacting to them properly can dramatically improve the quality of control. In short, a control system usually adopts models for the controlled environment, actuators, sensors and the controlling algorithms. Note that if a software system controls another software system, all the elements of a control system are implemented in software.

Subsections 3.4.1 to 3.4.3 present the important aspects of designing control systems.

### 3.4.1 Control Specifications and Algorithms

A large diversity of systems can be controlled, and therefore, controlled systems can have arbitrary complexity. Depending on the models associated with the controlled system, the analysis and design of a control system may include one or more of the problems that are identified in this chapter.

Assume for example that the quality of service parameters of a distributed system has to be controlled using the architecture given in Figure 4. Here, the model of the controlled system must express the actual quality of service parameters of the distributed system under consideration. The reference model must be expressive enough to represent the desired parameters. The controlling algorithms must have the knowledge of adapting the distributed system in such a way that the properties are adjusted in the right manner. The actuator must have the capability of executing the adjustments.

![Figure 4: Block diagram of a control system](image-url)
3.4.2 Sensing, Monitoring and Feedback

The control system must sense, monitor and control the controlled system. There is a meta-level relation between the controlled system and the control system. To effectively represent these meta relations, the underlying object-oriented model must provide a reflection mechanism.

Assume for example that the quality of service parameters of a middleware system has to be controlled. In this case, the controlled system is the middleware and the control system is responsible for the quality of service management. The quality of service parameters of a middleware can be adjusted, for example, by reconfiguring the middleware components. To be able to control the middleware, however, certain characteristics of the middleware must be measurable by the control system. In case of a closed middleware system, measuring the necessary parameters maybe too difficult or even impossible. In section 2.2.7, this was termed as the lack of support for reflection problem.

3.4.3 Coordinated Behavior

Complex and large control systems often consist of several distinct units such as controlling algorithms, actuators, sensors, and/or low-level sub-control systems. Depending on the architecture and/or controlling algorithms, the units of a control system coordinate together to keep the controlled system consistent with respect to its pre-defined specification.

As an example, consider again the implementation of the quality of service management system. Here, the measurements must be collected from the relevant and possibly distributed components. Similarly, adjustments must be realized on these components. This requires implementation of well-defined interaction mechanisms. If the interactions must be extended, the lack of support for coordinated behavior problem explained in section 3.3.1 can be experienced.

3.5 Distribution

A distributed computer system consists of multiple, cooperating, autonomous computer systems (nodes) that are interconnected by a communication network, and a middleware system to integrate such an interconnected system into a logical entity. In subsections 3.5.1 to 3.5.8, we present eight important aspects of distributed system design:
3.5.1 Remote Invocations

In a distributed system, components are scattered over the nodes of the system, and components on different nodes may need to communicate using the so-called remote invocations. The required semantics of remote invocations may differ according to the circumstances, although the remote procedure call model is used most frequently. In the remote procedure call model, the client component waits until the server component explicitly returns the result of the call. The remote procedure call semantics can be too restrictive for certain applications. In section 2.2.4, this was termed as the fixed message passing semantics problem.

3.5.2 Transparency

It is generally claimed that most of the activities within a distributed system must be transparent to the users/programmers of the system. The various possible sorts of transparency that can be added to distributed systems are summarized below:

Access Transparency: Whether or not there is any perceived difference of procedure in using a local resource or a remote resource.

Replication Transparency: Whether or not replication of components is visible.

Location Transparency: Whether or not access to (or control of) a resource is dependent on that resource of being at a particular location, held exactly one location, or distributed over several locations.

Failure Transparency: Whether or not failures that occur in system components affect the overall functioning of the system.

Although transparency makes it easy to write programs, there are some negative consequences, such as performance. In case of remote sharing, for example, performance can be negatively affected if topology information is not taken into account. Similarly, efficient and proper exception handling and increasing availability require more or less topology information. Moreover, in section 2.2.7, we discussed the negative affects of transparency to extending the services provided by the middleware system. Within this context two problems were discussed: First, the unmatched system functions problem, which can be experienced if the services of the middleware do not match the needs of the applications. Second, the lack of support for reflection problem, which can be experienced if the transparent middleware services are required to be modified or new services have to be introduced.
3.5.3 Resource Sharing

For sharing resources, generally a client-server model is used; a (server) component encapsulates the shared resource and serves requests from other (client) components. This model is sufficiently supported by the current component models.

3.5.4 Distributed Algorithms

Distributed algorithms are frequently required in distributed systems, not only for the implementation of middleware system level functions, but also for the realization of application level software systems. A distributed algorithm involves a number of components that exchange messages with each other.

In current component models, interaction code is likely to spread over all the participating components. In section 3.3.1, we discussed the difficulties in reusing and extending interaction code among components. This was referred to as the lack of support for coordinated behavior problem.

3.5.5 Distributed Concurrency Control

In a distributed system, data is partitioned and/or replicated over the nodes of the system, and commonly shared by multiple nodes. One of the major problems is to keep the data consistent. The most common mechanism to address this is the transaction mechanism. To guard data from inconsistencies in the presence of hardware or system failures, or when a certain sequence of actions have to be executed in an indivisible way, then the transaction mechanism can be adopted. A transaction is a sequence of events that are serializable and indivisible.

In the literature, many different transaction implementation techniques are presented. Implementations generally differ from each other with respect to their serialization and/or recovery characteristics. The suitability of an implementation technique may depend on the application, system and/or the data structure. It is therefore difficult to select a single implementation as a middleware service for all applications. In section 2.2.7, this was termed as the unmatched system functions problem.

If the application programmer requires choosing the implementation of middleware transaction services, and if the transaction services are transparent, the lack of support for reflection problem can be experienced. If the middleware has to automatically select the best transaction implementation in a transparent way, then this is a control system design problem as discussed in section 3.4. In this case, defining the right model for
the controlled middleware system and the right criteria for selecting the optimal transaction implementation can be a difficult task.

3.5.6 Recovery

Recovery is a facility for ensuring that no information is lost even in the occurrence of software and hardware failures. To cope with hardware failures, there must be a so-called stable storage, which is guaranteed to remain unaffected by failures. Recovery is usually part of the transaction mechanism; when a transaction succeeds, the new state is guaranteed to be saved, when the transaction is aborted half-way, the system is recovered to the state before the start of the transaction. The problems that designers may experience are similar to the ones of transactions.

3.5.7 Security

The security of an information system becomes more critical especially when the processing nodes are distributed. Access-control lists and capabilities are two well-known techniques [9]. Cryptographic techniques can be integrated with the system to reduce the vulnerability [14]. Security mechanisms must be superimposed upon other distributed system functions. Generally, this means introducing transparent services to the middleware. In section 2.2.7, this was defined as the lack of support for reflection problem.

3.5.8 Layered Communication Protocols

Distributed systems, in general, are structured in terms of layers. Functionally, each layer communicates with its peer-level layer, although physical data exchange occurs with the adjacent layers. In the section 2.2.7, the problems associated with designing layers were identified as the unmatched system functions and the lack of support for reflection problems.

3.6 Real-time

A real-time (RT) system is a system that is required to respond to external events within a predetermined time interval. RT systems are often classified into hard- and soft real-time systems. In hard-RT systems, violation of the time constraints will result in serious, possibly catastrophic damage to the system and its environment. Soft-RT systems aim at fulfilling the time constraints as much as possible. One class of soft-RT systems are statistical RT systems where average response times are required. For hard-
RT systems several analysis techniques have been developed to guarantee the timely behavior of the system under all circumstances.

In subsections 3.6.1 to 3.6.3, the important aspects of RT systems design are presented.

3.6.1 Real-time Specifications

In current component models, RT specifications may conflict with the composition mechanism. In case components with real-time behavior are required to be extended or modified through component composition, the designers may be obliged to redefine some or all of the features of the predefined components although this may be intuitively unnecessary. This is referred to as the composition versus RT specifications problem\(^9\).

RT specifications can also be defined for the coordinated actions of components, requiring an explicit representation of component coordination. The problem that is related to coordination was discussed in subsection 3.3.1 and was termed as the lack of support for coordinated behavior problem.

3.6.2 Dynamicity in RT Specifications

The current RT specification techniques lack flexibility in the association of RT specifications to components. In general, only the server component associates RT specifications with its operations, which are fixed during the life time of this component. Especially for soft-RT systems, it may be desirable to select an optimal RT specification for an operation from various alternatives. Moreover, client components are also not able to define time-intervals on their calls to server components. These are similar to the problems that were discussed in subsections 2.2.3 and 2.2.6 and were respectively termed as the multiple-views and lack of support for dynamic composition problems.

3.6.3 Temporal Behavior Analysis

Several algorithms for determining the temporal behavior of a RT system have been defined. As the problem itself is, at least NP-complete, the useful algorithms are heuristic algorithms determining an upper bound. There problems were discussed in the literature and termed as schedulability analysis. Efficient algorithms have to be defined for the analysis of RT component models. Other important aspect for soft-RT systems is exception handling in case the timing constraints can not be fulfilled.

\(^9\) This is similar to the real-time specifications inheritance anomaly problem [4]
4. CONCLUSIONS

Constructing software systems from components has several advantages such as managing complexity and run-time adaptability. There are, however, a number of obstacles that software engineers may experience while designing systems from components. In general, it is possible to classify these obstacles as lack of expression, decomposition and/or composition problems. In this chapter, these general problems are further specialized into 11 problems and are classified according to the aspects of domains of applications. By using the guidelines presented in this chapter, the software engineers may identify the potential problems first by identifying the domains of their applications, and then by considering the aspects of each domain.

Various techniques have been introduced in this book to overcome some of the obstacles presented in this chapter.

The composition and decomposition problems in design are addressed by the architecture design methods presented in chapters 3, 4 and 5.

The lack of expression is a general problem and addressed by parts 2 and 3 of this book. Every chapter addresses this problem within a specific problem context. For example chapter 6 presents a set of abstractions which are capable of expressing message-oriented architectures, chapter 7 presents a logic meta-programming language to capture architectural constraints, chapter 9 presents basic language mechanisms to express a large category of component composition mechanisms, etc. Chapters 10, 11 and 12 aim at enhancing the expression power of current languages by providing effective mechanisms for separation and composition of concerns.

The arbitrary composition problem is not directly addressed in this book. In our related research activity, we addressed this problem partially within the context of designing parser generators. We introduced the so-called grammar inheritance mechanism, in which sub-grammars may be inherited from the super-grammars [1]. This technique is especially suitable to reusing the grammars of specifications.

ACKNOWLEDGEMENTS

This research has been supported and funded by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs under the SENTER program, the Dutch Organization for
5. REFERENCES


10. E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Software. Addison Wesley, 1995.


GUIDELINES FOR IDENTIFYING OBSTACLES


APPENDIX DEFINITION OF THE OBSTACLES

- **Arbitrary composition**: was defined in subsection 3.1.1 as the difficulty in composing components if some or all aspects of components are generated from specifications and if the specifications cannot be composed by using the composition mechanism of the component model. This problem can be experienced mainly in designing application generators and constraint systems.

- **Composition**: was introduced in subsection 1.2 as the difficulty in creating new components by reusing simpler components. Effective composition demands two complementary characteristics from the component model. First, it requires as much as possible to reuse the existing components without modifying them. Second, the adopted composition mechanism must be suitable in utilizing the existing components in creating new components.

- **Composition versus RT specifications**: was defined in subsection 3.6.1 as the difficulty in reusing or extending RT specifications of components. This problem can be experienced in designing components with RT behavior.

- **Composition versus synchronization**: was defined in subsection 3.2.2 as the difficulty in reusing or extending synchronization code of components. This problem can be experienced in designing concurrent components with explicit synchronization.

- **Decomposition problem**: was introduced in subsection 1.2 as the difficulty in defining autonomous components in solving a given problem due to the complexity of the problem.

- **Excessive type declarations**: was defined in subsection 2.2.2 as a result of the necessity to declare a separate type module for every consistent composition of components.

- **Fixed message passing semantics**: was defined in subsection 2.2.4 as the difficulty of defining and reusing tailored message passing semantics of interacting components.
- **Lack of expression power**: was introduced in subsection 1.2 as the difficulty in expressing software directly by using the features of the adopted language.

- **Lack of support for coordinated behavior**: was introduced in subsection 3.3.1 as the difficulty of representing and reusing coordination among components especially if the coordination is implemented as multiple messages.

- **Lack of support for dynamic composition**: was introduced in subsection 2.2.6 as the difficulty of adapting composition structures (such as inheritance or delegation) at run-time.

- **Lack of support for reflection**: was introduced in subsection 2.2.7 as the difficulty of redefining the primitive or transparent features of the system.

- **Multiple views**: was introduced in subsection 2.2.3 as the difficulty of adapting the interface of a component based on its context.

- **Sharing behavior with state**: was introduced in subsection 2.2.6 as the difficulty of sharing a common behavior among components if the behavior is affected by a common state.

- **Unmatched system functions**: was introduced in subsection 2.2.7 as the mismatch of application needs and the functions provided by the system layer.