Contract-based Component System Design

Holger Giese
Institut für Informatik, Westfälische Wilhelms-Universität,
Einsteinstraße 62, 48149 Münster, GERMANY
gieseh@math.uni-muenster.de

Abstract

Component technology tries to solve many problems of todays software industry practice: the productivity and produced quality should be increased and a better infrastructure for maintenance of the products is promised. The integration of off-the-shelf components *to build customized products allows to source out the development of general purpose components. A crucial prerequisite for the intended scenario of component usage is their strong separation. Especially in a distributed environment, synchronization aspects are of great importance to identify a suitable architecture and to decide whether a component matches some requirements. The presented approach allows to model the synchronization aspect of* contracts *in a flexible manner including a whole spectrum of different degrees of preciseness from declaration of abstraction barriers to complete synchronization specifications describing the explicit behavior. The used Petri net based* OCoN *behavior specification formalism is structurally embedded in the UML and supports analysis and design of component systems.*

1. Introduction

The *complexity* of todays software projects is continuously growing and so does the need for sophisticated system analysis and design. Object-oriented analysis and design [5, 32, 22, 12] offers methods for analysis, design and implementation of systems in a seamless fashion. In contrast to *structured analysis* [13], the transition from design to implementation is more continuous. Traditionally, object-oriented techniques are used to specify fine grain structures using classes and their relations. Normally, one of the popular object-oriented programming languages, like C++, is chosen as target language. Often, the overall architecture or the coarse grain structure has been neglected or ignored at all. On the other hand, a dedicated design of a suitable *software architecture* [34] is often needed to improve software quality and to provide better maintainable products. But the hope that object technology can be used to establish systematic *reuse* has failed. The shift from objects to components reflects these additional requirements. A fixed architectural basis and system level mechanisms instead of programming language mechanisms are the crucial point to handle the described additional requirements and to achieve a more flexible notion for the composition of elements. *Component technology* [37] goes one step further in comparison with *object-orientation* as a language feature by decomposing an application or system into runtime elements, that can be build, analyzed, tested and maintained independently. The integration of available *off-the-shelf components* into applications and their combination can help to further improve productivity and decrease the time to market in the software industry.

A development method for component based applications and systems must be aware of additional problems. The design is further separated into *component design*, where a single independent shippable product for general use is the intention, and *component system design*, which considers combination and configuration of given components or the decomposition of a task into given and application specific components. *Component design* is restricted to isolated components having a fixed contract with the environment, while the *component system design* has to consider the coarse grain design and separation. The isolation between design and implementation of a component has to be supported by the architecture and a suitable separation. Otherwise the postulated component exchangeability and independence between component provider and component integrating products is not realistic. Both kinds of design problems have to face the resulting problems of late integration. The knowledge of common models for *software testing* using *module* and *integration testing* is not sufficient any more. The component notion of *quality* has to satisfy higher expectations, because the late integration phase is not available for testing any more. Thus, software components have to be more robust than usual applications. This additional demand for software quality may delay the development of a component market. The support for maintenance,

management and configuration has to be integrated into the component infrastructure.

Up to now, software products often provide isolated solutions for business or industry applications. Today software begins to interlink the different isolated information system structures. Interoperability, flexible data exchange and sharing as well as support for group work become essential requirements. Thus, *distribution* and *concurrency* are aspects, further generations of software have to manage.

The presented approach provides techniques and notations to tackle the additional requirements of component design. Structure and connections of *component systems* are specified using the structure description notations of the UML [31], the de-facto standard for object-oriented modeling. The common notion of interfaces is extended by a protocol to support contract-based design for components. Synchronization restrictions can further be specified in a flexible manner to describe dependencies between different contracts of the same component. Thus, the concrete interaction can be specified and architectural aspects become more obvious.

In the following section, several relevant characteristics of components and the available technology are discussed. Then, component synchronization and its impact is considered in section 3. The proposed approach is sketched in section 4 and its structural embedding into the UML is presented. An example in section 5 presents several different design decisions and their modeling with the approach. The article closes with some remarks on related work.

2. Component Notion

A general notion of a *component* should also include traditional component types like *libraries* or *modules*. Even when they do not support all characteristics of todays *off-the-shelf component* concepts, it is important to keep the basic concepts and their implications in mind. Besides the pure *off-the-shelf component* notion, there may exist several levels of component usage, which are of interest, too. Imported and exported types of a component are a relevant aspect as well as its connections with the environment. *Szyperski* [37, 38] defines a *component* as follows: *"A Software component is an unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*" For each interface a component has either contractually obligations or demands and thus at least some kind of informal contract for each of them exists.

The general description of a component consists of the component or subsystem itself and its imported or exported contracts. To make the contract notion more concrete, the approach clearly distinguishes between exported contracts, called *provided* and imported contracts, called *used*, w.r.t. a component. For provided contracts, the component has the obligation to serve it and for used ones the component may demand several contractual properties.

To figure out which aspects are of importance for a suitable contract notion, several characteristics of todays component concepts like linkage time and linkage typing are discussed next. Afterwards, the additional constraints for component design and the need to consider the *synchronization* between components is demonstrated.

A central characterization for component contracts or connections is the point in time when the connections are established (linkage). The traditional cases are *static linkage* of subcomponents at construction time of a program, *dynamic linkage* during the program startup or *runtime linkage* where running components are interconnected.

The typing of linked connections is also of considerable interest. For interprocess communication, *untyped interaction* based on streams, shared memory, etc. or even abstract synchronization with mutex or semaphores are used, while linking programs, modules and libraries often supports procedure typing on the compiler level. The case of *runtime* linkage is of special interest for todays component technology. Several levels of typing have been introduced. On the socket level, several services based on TCP connections have been standardized (ftp, nfs, http, etc.). To further support remote or local procedure call client/server interaction, common packet formats and integrated marshaling stubs (e.g. DCE [9]) have been used. These approaches still provide only a host-server abstraction, while object-oriented extensions introduce the object or interface notion to make service access points first class elements. CORBA [27] started from scratch 1989 as an initiative to build an interoperable object bus standard with suitable infrastructure. Its main antagonist is Microsoft's DCOM [10] which is a step by step extension of COM (*component object model*, formerly named *common object model*). These approaches allow to send and distribute interface references as usual parameter values. Java RMI [36] further extends this development by also supporting the *object per value discipline* within its *remote method invocation* mechanism. CORBA, DCOM and Java RMI are enabling technologies which provide typed component linkage at runtime. To discuss this development, the relevant aspects for runtime linkage are of interest.

When untyped basis mechanisms like TCP sockets are the linkage mechanism, a suitable connection has to be described by defining all valid packet formats and an agreement on the protocol built upon the packet formats. When abstracting from the basic TCP protocol steps to establish a connection, often simple *stateless* protocols like the common basic HTTP [3] protocol, which uses the request/response scheme, are used. These protocols provide

a high degree of independence, which is often useful in a distributed environment.

The further improved typing using client/server approaches manages the error prone encoding of packets and provides the higher level concept of a *remote procedure call*. In general this basic scheme of interaction does not make an explicit interaction protocol obsolete. Client/server systems often provide a *stateless* protocol, e.g., NFS [35] is based on a standardized *remote procedure call* mechanism and is a *stateless* and *idempotent* protocol to handle connection aborts and re-transmissions. It is remarkable, that common network based services like NFS avoid any complex interaction with third components and thus build final leaves in the component tree or directed acyclic graph.

The CORBA or DCOM object bus approaches provide the illusion of a virtual object space, where interfaces instead of hosts abstract from physical locations. These references can be further distributed to make them available to other clients. But, their typing notion is still restricted to the syntactical interface aspect. Complex protocols and their impact on a correct cooperation are not considered. The object bus *interface* concept does essentially combine data and behavior by applying the object metaphor. Thus, the resulting protocols might not always remain *stateless* as common for the design of services like NFS or HTTP.

Traditionally, the basic mechanisms used for component reuse and *static linkage* are the libraries which provide a *procedural abstraction* with strict acyclic depending layers. The explicit sharing of resources is avoided where possible. The common components for *dynamic linkage* are either named *shared library* or *dynamic link library* (dll). They support a perfect separation for the using clients and provide the perfect illusion of *exclusive* usage, too. Also, a layered structure from the operating system API up to domain specific or more comfortable libraries is common. Both scenarios provide contracts in an exclusive fashion and abstract from code or data sharing. The *off-the-shelf component* concept in contrast is intended to support arbitrary structures, has to be able to allow more sophisticated interaction concepts like *callbacks*. Also the restriction to *stateless* protocols is often not possible.

Besides the basic object bus infrastructure and a communication mechanism, component based development requires further aspects. DCOM supports components with its ActiveX or DNA architecture as well as Java *Enterprise JavaBeans* (EJB) [23]. A specification of a component model for CORBA is under development (see [1]). These component models improve the basic object bus technology by specifying interfaces for several basic component management aspects and support for component *life cycles*. But besides these technical solutions to obtain interoperable runtime components, the necessary contract specification is neglected. In contrast to the former definition for *components* which emphasizes the *contract principle* [24] as essential aspect of any component technology, the specification of contracts in practice is not supported by any object bus technology. Instead, the handling of interface contracts is assumed to take place in additional specification documents and additional features like unique interface version numbers are used to achieve consistency.

3. Component Synchronization

Szyperski [37] identifies another serious problem occurring when *callbacks* are used. He demands to specify re-entrance conditions to cover these problems, but *re-entrance* is only a special case of the more general question how components may *synchronize*. When *state based protocols* are considered and concurrency is present, a general treatment of *synchronization* aspects is needed.

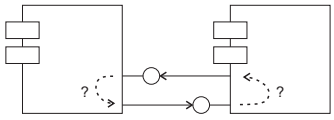

Figure 1. Example structure for a callback

The structural situation of a *callback* is visualized in figure 1. The provided and used contracts build a cyclic dependency and thus the classical procedural abstraction fails and instead *synchronization* aspects have to be considered, too. In classical layered hierarchical systems, *callbacks* against the hierarchy called *up-call* [11] cause several problems and enforce the library designer to provide a consistent library state even during such calls.

Using *thread-safe* objects does not ensure systems which are also *re-entrance safe*. Phenomena like *self-recursion* and *re-entrance patterns* additionally lead to deadlocks (so called *self-infected deadlocks* [8]). But even in simple cases, the system malfunction may be caused by synchronization effects. Consider, for example, the case of a component with a single thread of control. When it calls another component via a remote procedure, it is blocked until the request is processed and thus any callback is blocked. If the called component waits for the callback to fulfill the request, at least the first component is totally blocked forever due to a resource conflict concerning its single thread. For components in a distributed environment the situation becomes even more complex and the system operation may critically depend on the request scheduling strategy of the implementation.

Object-oriented type structures often contain cycles (recursive data types), but traditional object-oriented systems were not concurrent, and, hence, this aspect has often been ignored. In the case of multiple threads or concurrency in

general, the *synchronization* becomes even more important. Consider as an example the classical recursive defined directory class. A first version may not support file links and thus *cycles* are excluded. But when links are also considered, a possibly cyclic structure is described. Common realizations like file systems reflect this by extending related tools to prevent infinite processing (e.g., Unix find command). Directory like structures in distributed systems are found in an Internet name server. There, an asynchronous update scheme is used and thus no update request can lead to infinite processing, because only the local cache content is propagated. The CORBA name service [28] also provides the directory access in such a way that any direct usage of related directories is excluded. Instead, the client has to traverse the structure on its own. By avoiding any global operation, the *synchronization* and *termination* problems can be excluded, but the complexity is left to the clients.

Object protocols with states or some kind of *life cycle* are common in object-oriented systems. The possible processing orders are specified, for example, by using Harel statecharts [19] in OMT [32] and path expressions in FUSION [12]. The life cycle or protocol describes the possible *non uniform service availability* provided by the object.

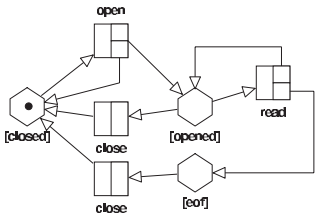

Figure 2. The protocol of a read file handle

Consider for example the read file handle protocol presented in figure 2. Reading data chunks is only supported after open the file. Than, data chunks can be read until the end of file ([eof]) is reached. When the file is closed (close), again no read operation is available. The OCoN notation [17] is used to describe the resulting state changes and the available operations in each state as well as the resulting state. Hexagons represent possible states and actions consisting of a call and return step with possible multiple return alternatives are represented by squares.

The combination of components during the *component system design* is different from combining and designing classes during the fine grain object-oriented design. The object-oriented techniques support encapsulation by private and public access to classes. This style does not fulfill the additional requirements for separation. CORBA, DCOM and Java RMI use interfaces to decouple specification and implementation, but additional information necessary to ensure a correct integration is missing. As demonstrated, the syntactical interface typing does not cover all relevant aspects for component composition. It determines all message formats of a protocol by defining a standard encoding, but it does not describe which processing order is needed. Only interfaces with *stateless* protocol in situations without *re-entrance* and *cyclic* structures are covered. Suggested trace-based extensions [29, 30, 25] can exclude the occurrence of message not understood errors, but fail to consider synchronization effects.

Following *Szyperski* [37], a *contract* should contain a functional specification usually given by pre and post conditions and non-functional requirements often named *service-level* or *quality of service* containing aspects like *availability*, *throughput*, *latency* and *capacity*. As demonstrated above, synchronization is another important aspect, but behavior modeling is an inherent complex problem. *Beugnard* et al. [4] present a contract hierarchy that systematically distinguishes *basic contracts* which represent the common interface notion, *behavioral contracts* that provide pre and postconditions, *synchronization contracts* for several request synchronization policies and *quality of service contracts* covering aspects like *availability*, *throughput*, *latency* etc.

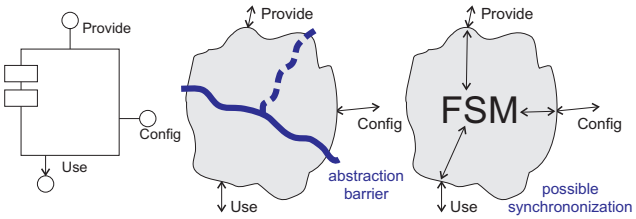

Figure 3. Abstraction barrier

When the interaction of arbitrary structured systems of components is considered, the *synchronization* is of crucial importance. The abstraction assumed for a component (figure 3, left-hand-side) is usually characterized by an *abstraction barrier* (middle), while the real synchronization (right-hand-side) does not respect it. A formalism like a finite state machine (FSM) has to be used to describe the behavior aspect using states and transitions.

When components are connected, their external synchronization specification has to be combined to obtain the resulting behavior. This explicit combination results already for very restricted system models to serious problems. For the *finite state machine* formalism chosen in figure 3 and every more expressive formalism, the state space grows exponential, known as the *state explosion problem* [39] for system analysis. Formal approaches to system verification and validation try to overcome this problem, but the explicit modeling of interaction includes several aspects like synchronization distances which contradict this.

But this problem is also of crucial impact for the system design concerning *change impact*. The exponential growing model sizes coincide with an exponential growing number of implicit implementation dependencies. The transitive nature of synchronization for two connected component systems causes this problem. Thus, to change a component implementation may influence every other implicitly connected one. This effect can be prevented by restricting the general interaction and structure as done in the case of libraries. The approach proposed here avoids the demonstrated problems by using the *abstraction barrier*, visualized in figure 3, when suitable and the explicit specification of synchronization if needed. The synchronization can even be described with different levels of preciseness. This also improves the resulting situation for the design of the system. Callbacks or even cyclic structures introduce complex interaction dependencies and the concrete external behavior has to be specified very early in the design. Otherwise both involved components can not be further considered in isolation. Thus, the proposed approach does combine the improvement for the analysis and design as well as formal modeling by supporting *abstraction barriers* as design principle and as mechanisms to make a formal analysis feasible.

4. Contract-based Design

The presented approach emphasizes contract-based design to improve separation using *synchronization contracts*, extends the contract notion to cover bilateral interaction in a manner which still leads to unilateral dependencies as well as the explicit design concerning the component contract structures and cycles.

The formalism of the OCoN approach [41, 16, 17, 18] for *seamless* object-oriented behavior modeling is used also to cover the behavioral aspects of components. OCoNs (Object Coordination Nets) formally defined in [15], a special form of *Petri nets* [6], are used to describe the possible protocol interactions in a visual manner. These nets specify the intended interaction and allow to describe procedure-call and message-passing oriented interaction within one formalism. In object-oriented design practice, behavioral aspects are often only considered when already implementing the system. Thus, *synchronization* aspects have not been well or completely documented during design phase and the needed information concerning the synchronization with the environment are usually not available during the design. In contrast, the OCoN approach supports the modeling of synchronization and coordination aspects during the design. The resulting component specification can be extracted from the component design and not from the implementation. On the other hand, if contracts are specified during the decomposition of the system, new general purpose or application specific component specifications including their synchronization behavior are obtained.

By emphasizing the contract idea, the using and providing components have to agree on each contract. The following parts of a contract description can be distinguished: a *protocol* describing the provided coordination sequences and a *functional specification* given by pre and post condition formulas. While the protocol is already considered during the design phase, the pre and post conditions can only be used for *verification* and runtime checks. Thus, the approach concentrates on the protocol aspect, which can be supported by tools for restricted models.

The *contract* notion is of central relevance for the design process. *Nierstrasz* [26] proposes to add a finite state machine to an object interface to build *regular types*. This approach is extended by also integrating the occurrence of return alternatives and *spontaneous* contract behavior into the protocol specification. Thus, instead of error prone direct callback designs, an encoding into the protocol states and spontaneous behavior can be used in most cases. Thus, a client obtains an unilateral contract which does not contain any obligations for the client side. The only exception is that the replies for pending operation calls must be at least buffered by the client to exclude the blocking of the called component.

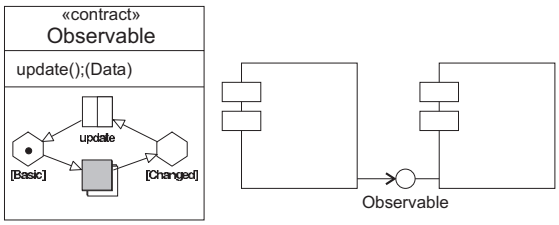

Figure 4. An Observable contract

For an example consider the Observable contract presented in figure 4, which provides a solution for the observer pattern that is still unilateral concerning the synchronization and typing dependencies. An additional arbitrary state change for the observable contract is modeled using a *quiescent* step [15]. Its occurrence is neither determined nor guaranteed. The client may observe the state change and do an update as needed. The still unilateral contract thus can be used to avoid cyclic dependencies as introduced by the general scheme of a callback presented in figure 1. Thus, the approach integrates bilateral interaction into an unilateral contract and can further provide maximal degree of flexibility for the using side (client).

The component behavior can be specified in an operational fashion also using the OCoN approach (see [17]). Thus, the contract protocol can be used to simulate parts of a system in an abstract fashion by representing the environment by its contract protocols. But such explicit design

including several component internal aspects is not suitable in general. A more abstract and implicit solution is needed.

Traditionally, architectural aspects are often neglected in object-oriented system design. By considering connections like *connectors* to be a kind of first class elements of an architecture (see [2]), it is achieved that the architectural aspect is specified adequately. In order to apply the concept of connectors and the *contract principle*, an UML <<contract>> stereotype containing an interface describing a set of interaction steps and a protocol description specifying the supported interaction orders is introduced. A single contract is unilateral and describes what behavior one interface of a component assures and how another component can interact with it. For a more detailed description see [18, 15].

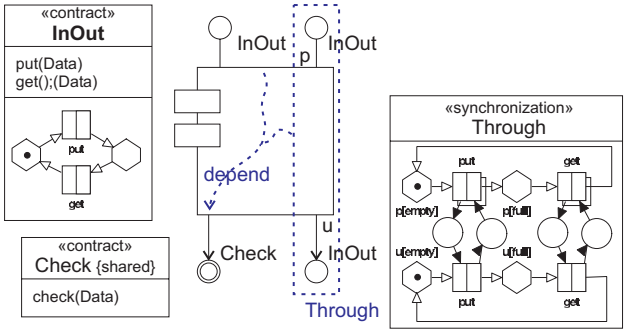

Figure 5. Structural extensions to the UML

The contract is used to describe the combination of an interface and a *protocol net*. In contrast to the UML interface notion, the contracts are instances and the relations among them are explicitly modeled as presented in figure 8.

There are two distinct kinds of contracts, *exclusive* and *shared* ones. This technical distinction for contracts is in conformance with the ISO Open Distributed Processing model [21], where implicit and explicit bound objects are distinguished. The *exclusive* contracts are interpreted as explicit bound objects while *shared* contracts fit to implicit bound objects (cf. [25]).

For an *exclusive contract*, the interface circle symbol is used as a shortcut (see figure 5) and for all usage connections an implicit xor and client side cardinality 1 is assumed and thus omitted, too. For the connection to the providing component only the number of served instances is of interest. Each contract is served by exactly one component and thus the component side cardinality is omitted. For *shared contracts* also sharing by multiple clients is allowed and thus the usual cardinality annotations can be used for connections to the clients. A circle with double border is used as shortcut. The annotations for connections to the providing component are the same as for the *exclusive* case.

To provide the demanded component specification, the synchronization of *provided* and *used* contracts has to be specified. Two situations for contracts are further distinguished. Either they are *simple* and their guaranteed operations are not restricted or an additional <<synchronization>> stereotype is used to further restrict the protocol by introducing synchronizations with other provided or used contracts of the same component (see figure 5). These synchronization declarations are added to each component type and are additionally visualized using a dashed box around every covered contract. Each contract can at most take place in one such synchronization and thus the dashed rectangles of one component can not share any contract declaration. The Through synchronization presented in figure 5 describes how the put and get operations of the provided contract InOut p are mapped to the used contracts u of the same type. The actions with a shadow describe the processing of incoming requests for the p contract while the usual actions specify the requested operations for contract u. The synchronization is described using untyped places (circles) and additional pre and post condition arcs. Each requested put or get is forwarded from p to u and the return is processed vice versa.

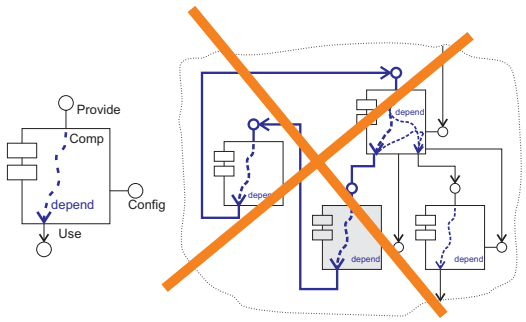

Figure 6. Embedding and depend relation

Besides these explicit synchronization descriptions, also an implicit description using a *synchronization dependency* relation depend ($\rightarrow$) is supported by the approach. The synchronization is not explicit described and instead any arbitrary but valid usage of *used depending* contracts and no *synchronization* with *used independent* (not connected) contracts is assumed. If neither an explicit specification nor such an explicit relation is given, simply the worst case of a full dependency relation is assumed. This way the traditional abstraction barrier between exported provide contracts and imported use contracts can be used. A behavior cover is build by all possible implementations for each provided contract that synchronizes at most with all used contracts, the provided contract depends on ($\rightarrow$). Each correct implementation has to respect this behavioral cover. Each orthogonal line to all depend arcs builds a suitable abstraction barrier. But the provided abstraction is not valid in gen-

eral. The transitive extension of all local depend annotations has to be acyclic to make the assumed abstraction a correct one.

As demonstrated in figure 6, the depend relation restricts the valid embedding of a component. But this way, an explicit and complete synchronization specification can be avoided. The provided contract Config is used to configure the component and thus should be implemented in a fashion that does not rely on the used Use contract. In contrast, the Provide contract will rely on the correct responses of the Use contract. This dependency is specified by defining a component specific depend relation "$\rightarrow$" using dashed arcs. Thus, the possible component behavior is already restricted concerning the possible synchronization dependencies, but still a whole bunch of possible internal component behaviors are suitable solutions.

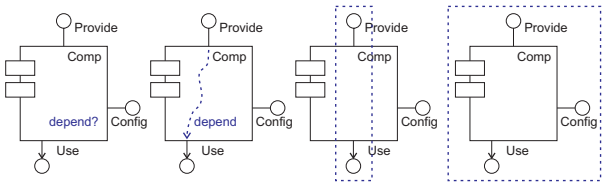

Figure 7. Spectrum of possible specifications

The provided mechanisms for contract specification allow to specify the contract behavior and their synchronization with several levels of granularity as presented in figure 7. Starting during the component system design, the relation may be left unspecified and thus a complete depend relation connection of each provided interface with all *used* ones is assumed. When further knowledge about the separation and wanted parallel availability for interfaces is given, a refined view by specifying an explicit depend relation is possible. If the planned embedding enforces the explicit modeling of synchronization aspects concerning a subset of the component contracts, this can be done using a <<synchronization>> stereotype. Now, *slices* of the component behavior can be specified in an independent fashion. A complete behavior description is also possible using a single synchronization element that covers all provided and used contracts. Thus, also a behavior description enclosing the whole behavior as described in figure 3 (right-hand-side), is possible. The provided spectrum allows to specify the behavior in the adequate level of granularity during the decomposition of the design. For already fixed components, e.g. off-the-shelf, provided by others, a specification of suitable preciseness may be chosen and can be used to embed them into a design.

To provide a sound framework to handle component protocols and their synchronization, the correct behavioral preorder describing a correct abstraction or refinement is needed. The given synchronization protocols can be compared as labeled nets by considering the label occurrences. The underlying formalism to determine a valid refinement or abstraction step concerning the component synchronization is then *reduction* [7], which is the coarsest relation w.r.t. preserving deadlock-freeness (see [40]). The symbol $\sqsubseteq$ is used where $A \sqsubseteq B$ states that A is a valid refinement of B. The abstraction from finite internal interaction can be used to replace a synchronization combination by a single more abstract version where the double covered contracts are omitted (see for example figure 10).

5. Example

To give an example, the common pipeline processing of a compiler is considered. The structure may consist of a pre-processor phase for macro expansion as well as a compiler with lexical analysis, syntactical analysis, semantical analysis, code generation and assembly stage. This software architecture style provides a high degree of flexibility and distinct stages may be exchanged on demand, e.g., the assembly stage to adjust the compiler to a certain hardware. By specifying the data format for each stage transition, each stage does only communicate with its predecessor and successor and thus the coupling is minimized. In order to reduce the example complexity the same general interface for each stage is assumed. Two solutions for a general pipeline structure are presented.

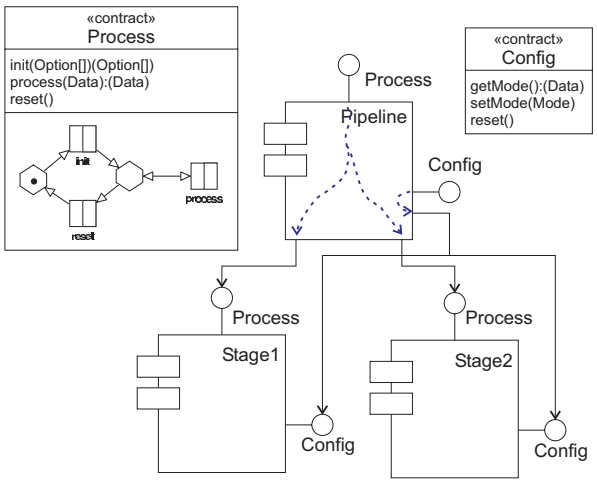

Figure 8. Pipeline with coordinator

The first way to build a pipeline structure is to use filter or transformer functionality in form of *remote procedure call*s. The resulting components are very flexible concerning further embedding and can be used for arbitrary requests or in a coordinated fashion like the pipeline structure. A

pipeline can be build using a specific coordinator component, as demonstrated in figure 8 for the trivial case of two stages. It is remarkable that the structure does not reflect the pipeline and instead the usage relationship from the coordinator component to each stage is made explicit.

This flexibility is a suitable reason to choose this solution, while efficiency reasons make this solution suboptimal. The applied *remote procedure call* interaction and the *central* coordination using an additional coordination component results in doubling the communication and a possible *bottleneck* for long pipelines. The bottleneck can be avoided by using a tree like coordinator structure which further increases the communication overhead. Each node in the tree provides the same synchronization type as a leaf and abstracts from the inner pipeline structure.

A more efficient design can be build by avoiding the overhead of moving the data to the pipeline coordination component and vice versa. Instead, the stage components are directly connected and each component has to provide an input and output stream (see figure 10).

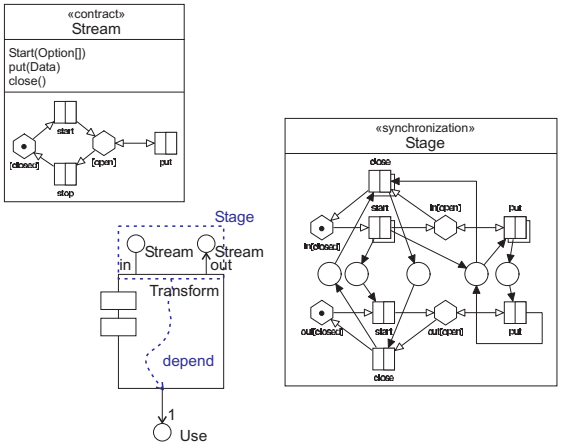

Figure 9. Transform component

The general scheme of independent provided contracts and a simple depend relation is not sufficient any more. Instead, the complex contract notion specifying the combined behavior of sets of provided and used contracts is needed. For the provided contract in and the used contract out a specific combined behavior is described in figure 9. This complex contract behavior is realized using a <<synchronization>> Stage, which synchronizes the in and out contracts of a single component. An incoming start request for in asynchronously triggers a start request for out and an internal place is initialized with a token. Afterwards each put is forwarded and returns immediately when no old put request for out is pending. Closing the contract is delayed until the out contract confirms the close request. Note, that the actions are a shorthand notation for two steps, a call and a corresponding return step. For example, the out.close post condition is a pre condition of the in.close return step. The resulting processing specification describes the explicit buffering behavior and thus even cyclic pipelines like ring structures may be build.

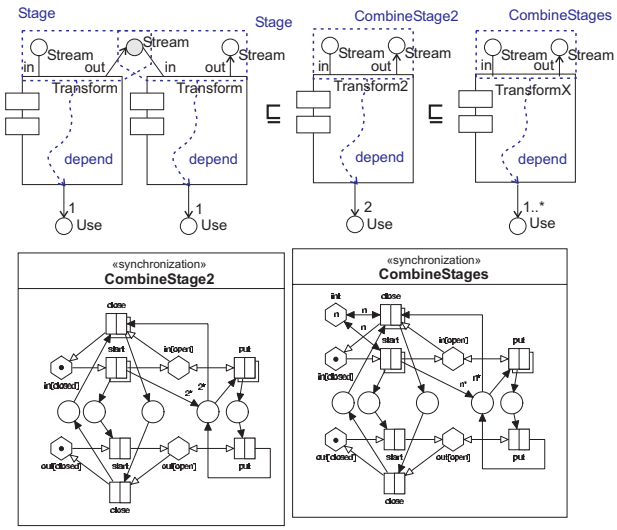

Figure 10. Pipeline of Transform components

For this non hierarchical structure it can also be abstracted from two or more stages by combining their complex contracts and abstracting from their inner communication. See figure 10 for the resulting behavioral cover of two synchronously connected Stage synchronization restrictions. The resulting common behavior of two stages has to describe the internal buffering in a concrete fashion. When an arbitrary but non-determined internal buffering like described by the second abstraction, is used, the resulting behavioral cover can be combined and used only in a restricted way. Consider a cyclic pipeline case and combine n of these abstract stage components. For a secure processing at most $n-1$ data packages can be inserted. Otherwise the cycling might be blocked and thus n or more packages may not work. For such ring like structures abstracting from the buffering effect is not always useful. When abstracting from the buffer depth the information is lost and not available. The ring structure will only work if at least one buffer element is still empty. If all buffer capacities are exhausted, each stage will be blocked by the next one and no progress is possible any more. Thus, abstracting from the buffering depth may be not appropriate.

The described two behaviors CombinedStage2 and the more abstract version CombinedStages are valid abstractions for the behavior preorder ($\sqsubseteq$). Their nets describe the explicit buffering where $x*$ denotes x token and the resource of type int is initially filled with an arbitrary value

n. This process of abstraction can also be reverted and the more concrete version can be considered as a *refinement* of the coarser description. Hence, a pipeline build by several CombinedStages can be refined to a CombinedStage2 or Stage.

6. Related Work

The presented approach extends the concept of *regular types* of *Nierstrasz* [26]. In contrast to this reduction-based approach, do trace-based notions [30, 25] not consider synchronization effects and exclude only message not understood errors. To reduce cycles in the usage graph, the unilateral contract notion is extended to include bilateral interaction. This way most of the error prone *callback* handling can be handled in a more suitable fashion. The presented protocol formalism additionally covers distinct synchronization with the request replies, while *Nierstrasz* work is restricted to request acceptance. This way oracle like requests and the influence of distinguished replies on the resulting protocol state can be incorporated, too. The concept of explicit contract synchronizations and an implicit contract depend relation further extend the framework towards a flexible specification tool for component synchronization.

The integration into the analysis and design level instead of the programming language or a formal calculus context is another distinction. The approach allows to consider synchronization and protocol aspects, which are of great importance for the architecture design, already during the analysis and design. It supports the specification for incomplete system, refinement and explorative design evaluations by simulation.

Holland et al. [20] suggest a contract notion that abstracts from performance and resource consumption aspects and includes *safety* and *progress conditions*, which are needed to predict the component behavior from a clients perspective. So called *type obligations* demand abstract attributes and interface aspects for each participant while *causal obligations* describe the ordered sequences for actions and their effect on the attributes. The CATALYSIS [14] approach emphasizes a pre and post condition concept but also contains a comparable concept as extension and suggests statecharts or sequence expressions to specify the order of internal called actions called *raised actions*. The general concept to describe object behavior for a group of objects is promising, but the resulting system is more suitable for frameworks. The superposition of such interaction concepts is not always conflict free and the pre and post conditions or invariants make an automatic tool support impossible. In the area of object-oriented design for real-time systems, the ROOM [33] method also uses protocols defined for a group of objects and signal based protocol roles called ports as connectors. In contrast to the presented approach, the protocol is used to describe the bilateral signal exchange and no notion for behavioral abstraction is considered. The structural description techniques of the UML are used for the structural part of architecture descriptions. For behavior specification, the OCoN approach is used, because it provides a *seamless* integration of used or provided contracts (see [17]). The behavior description techniques of the UML are not capable of these aspects. For a comparison between the OCoN approach and the behavior formalisms of the UML see [18]. Remarks concerning the great variety of other proposed object-oriented Petri net notations can be found in [17].

7. Conclusion

The presented approach provides mechanisms to achieve a higher degree of independence, to exclude implicit implementation dependencies and make requirements and the provided behavior of components more concrete. The OCoN formalism together with the presented extensions provides a suitable framework for the described component design techniques. An external behavioral specification technique using *synchronization slices* and a *declarative depend* relation for implicit synchronization specification is presented. From the perspective of formal model specification, a *seamless* transition from totally separated contracts and a dependency relation over slices of partial behavior specifications to a complete external behavior specification is supported. The unilateral contracts with possibly shared protocols refining the connector concept allow to analyze the quality of an architecture concerning decomposition on well established object-oriented knowledge. The formal description of interaction properties by object coordination nets allows the analysis of behavioral properties and possible interaction scenarios can be simulated and visualized.

References

[1] Corba Components. (final submission), OMG TC Document orbos/99-02-05, Mar. 1999.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connections. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[3] T. Berners-Lee, R. Fielding, U. Irvine, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. IESG, May 1996. RFC 1945.

[4] A. Beugnard, J.-M. Jezequel, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.

[5] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Menlo Park CA, 1993. (Second Edition).

[6] W. Brauer, W. Reisig, and G. Rozenberg [eds]. *Petri Nets: Central Models (part I)/Applications (part II)*, LNCS 254/255. Springer Verlag, Berlin, 1987.

[7] E. Brinksma, G. Scollo, and C. Steenbergen. Lotos Specifications, Their Implementations and Their Tests. In G. Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing and Verification*, pages 349–360. North Holland, 1987.

[8] L. Cardelli. Obliq: A Language with Distributed Scope. Techreport, DEC System Research Center, Palo Alto, 1994. DEC SRC Research Report 122.

[9] D. Chappell. The OSF Distrbuted Computing Environment (DCE). In R. Khanna, editor, *Distributed Computing: Implementations and Managment Strategies*. Prentice Hall, 1994.

[10] D. Chappell. *Understanding ActiveX and OLE - A Guide for Developers and Managers*. Microsoft Press, 1996.

[11] D. D. Clark. Structuring a system using up-calls, Proceedings 10th ACM Symposium on Operating System Principles. *ACM Operating System Reviews*, 19(5):171–180, 1985.

[12] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[13] T. DeMarco. *Structured Analysis and System Specification*. Engelwood Cliffs: Yourdon Press, 1978.

[14] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Prentice-Hall, 1999.

[15] H. Giese. Object Coordination Nets 2.0 – Semantics Specification. Techreport, University Münster, Computer Science, May 1999. 15/99-I.

[16] H. Giese, J. Graf, and G. Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. pages 107–116, July 1998. Proc. Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'98), Kyoto, Japan.

[17] H. Giese, J. Graf, and G. Wirtz. Seamless Visual Object-Oriented Behavior Modeling for Distributed Software Systems. In *IEEE Symposium On Visual Languages, Tokyo, Japan*, Sept. 1999.

[18] H. Giese, J. Graf, and G. Wirtz. Closing the Gap Between Object-Oriented Modeling of Structure and Behavior. In *The Second International Conference on The Unified Modeling Language Fort Collins, Colorado, USA*, Oct. 1999.

[19] D. Harel. Statecharts: A Visual Formalism for complex systems. *Science of Computer Programming*, 3(8):231–274, 1987.

[20] I. M. Holland. Specyfying reusable components using contracts. In L. Madsen, editor, *ECOOP'92- Object-Oriented Programming, 6th European Conference, Uetrecht, Netherlands*, LNCS 615, pages 287–308. Springer Verlag, June 1992.

[21] ISO/IEC. *Open Distributed Processing Reference Model - parts 1,2,3,4*, 1995. ISO 10746-1,2,3,4 or ITU-T X.901,2,3,4.

[22] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: a use case driven approach*. Addison-Wesley, 1992.

[23] V. Matena and M. Hapner. *Enterprise JavaBeans^{TM} Specification*. Sun Microsystems, May 1999. Version 1.1, Public Draft.

[24] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. 2nd edition.

[25] E. Najm, A. Numour, and J.-B. Stefani. Infinite Types for Distributed Object Interfaces. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. 3rd Int. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS), February 15-18, 1999, Florence, Italy*, pages 353–369. Kluewer Academic Publischers, 1999.

[26] O. Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[27] Object Management Group. *The Common Object Request Broker: Architecture and Specification, CORBA/IIOP 2.2 Specification*, Feb. 1998. Revision 2.2: OMG Technical Document formal/98-07-01.

[28] Object Managment Group. *CORBAservices: Common Object Services Specification*, Nov. 1997. Revision 2.1.

[29] C. Peter and F. Puntigam. Coordination of CORBA Objects with Process Types. In N. Davies, K. Raymond, and J. Seitz, editors, *Middleware 98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer Verlag, 1998. (online report).

[30] F. Puntigam. Coordination Requirements Expressed in Types for Active Objects. In M. Aksit and S. Matsuoka, editors, *ECOOP'97- Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland*, LNCS 1241, pages 367–388. Springer Verlag, June 1997.

[31] Rational Software Corporation. *Unified Modelling Language 1.1*, Sept. 1997.

[32] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[33] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[34] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging Discipline*. Prentice Hall, 1996.

[35] Sun Microsystems. *NFS: Network File System, Version 3 Protocol Specifcation*, Feb. 1994.

[36] Sun Microsystems Inc. *Java^{TM} Remote Method Invocation Specification*, Oct. 1998. Revision 1.50, JDK 1.2.

[37] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[38] C. Szyperski and C. Pfister. Workshop on Component-Oriented Programming, Summery. In M. Mülhäuser, editor, *Special Issues in Object-Oriented Programming – ECOOP'96 Workshop Reader*. dpunkt, Heidelberg, 1997.

[39] A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets 1998 Part I*, LNCS 1491, pages 429–528. Springer Verlag, 1998.

[40] W. Vogler. Failures Semantics and Deadlocking of Modular Petri Nets. *Acta Informatica*, 26:333–348, 1989.

[41] G. Wirtz, J. Graf, and H. Giese. Ruling the Behavior of Distributed Software Components. In H. R. Arabnia, editor, *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada*, July 1997.