

Towards Ruling Component-Based Distributed Systems with Role-Based Modeling and Cross-Cutting Aspects

Holger Giese
Software Engineering Group
Department of Mathematics and Computer Science
University of Paderborn
Warburger Str. 100, 330.65 Paderborn
Germany
hg@uni-paderborn.de

Abstract

Today component-based distributed systems and integration of beforehand isolated information system solutions is one focus of IT industry activities. Such projects are often characterized by an overwhelming complexity due to heterogeneity w.r.t. technical and conceptual aspects. Therefore, we suggest to employ advanced separation of concern techniques to reduce the complexity. Role-based modeling combined with cross-cutting aspects as two complementary approaches for advanced separation of concerns are considered. A concept for component-based distributed system design and architecture is discussed which supports both in a common framework. The approach emphasize the contract principle in form of role-based modeling. The support for secure runtime binding of contracts and explicit contexts further permit to consider the case of cross-cutting aspects even for dynamic binding and open systems.

1. Introduction

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. *Component technology* [15, 1] offer a promising technology to tackle such distributed systems. It goes one step further in comparison to *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 should help to further improve productivity and decrease the time to market in the software industry.

The fundamental principles to break complexity in software engineering such as *separation of concerns*

[12] are employed in component-based system right like in programming language environments. Therefore the common notion of modules with interfaces leads also to *one dominant dimension of separation* [16] and thus, depending upon design decisions, other design aspects become *cross-cutting aspects* [7]. *Multi-dimensional separation of concerns* [16, 5] has therefore been proposed to support the separated handling of different aspects even when one dominant decomposition is given. In the context of distributed systems however we have to reevaluate the techniques proposed for programming languages.

2. The Approach

We use the UML [11] component diagrams as visualized in Figure 1 while instead of the fuzzy UML definition we define a component to be a unit for independent deployment and third party composition with contractually guaranteed and required behavior. It is further useful to clearly distinguish between realized contracts, called *provided* and needed contracts, called *used*, w.r.t. a component.

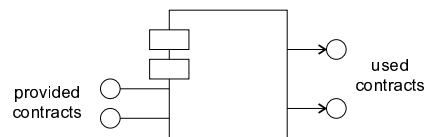


Figure 1. Notion of a component

The *typed component systems* (TYCS) [2] approach provides a basis for component-based systems which incorporates concurrency and distribution aspects. The following parts of a contract description are further distinguished: a *protocol* describing the provided coordination sequences where required and a *functional specification* given by pre- and post-condition formulas. 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 used. For a more detailed description see [4].

A component can play multiple roles (see OORAM [13]) at the same time. Therefore, *multiple interfaces* [17] can be employed to improve *separation* by partitioning the feature sets into independent contracts to achieve a more flexible design and reduce the size of each contract protocol. Recent work [6] suggests to realize role-based models with AOP. This fundamental relation is also exploited in the approach on the level of system design.

For the functional specification and safety properties the modular construction of contract-based components is straight forward. The progress for multiple contracts, however, cannot always be guaranteed by the realizing component independently of the component environment and the interplay with other components. A local contract view is therefore only valid in a strictly hierarchical architecture and when components serve their contracts in a fair manner. If more general forms of architectures are considered the relation between provided and used contracts of components cannot be ignored. In contrast to safety properties it is problematic to ensure liveness properties such as progress for arbitrary connected components. We therefore generalize the idea of [8] to support separation for progress properties even for non-layered structures employing explicit contract dependencies for a given set of provided and used contracts. Therefore, besides the explicit synchronization descriptions with protocols for each contract, also an implicit description using a *synchronization dependency* relation $\text{depend} (\rightarrow)$ is supported to address the overall component synchronization behavior. The synchronization is not explicitly described and instead any arbitrary but valid usage of *used depending* contracts and no *synchronization* with *used independent* (not connected) contracts is assumed.

A behavior cover is build by all possible implementations for that serve all provided contracts in a fair manner and serving a contract is at most blocked as long as its used contracts, the provided contracts it depends on (\rightarrow), are not served. Each correct implementation has to respect this behavioral cover. Each orthogonal line to all depend arcs builds a suitable abstraction barrier. However the provided abstraction is not valid in general. The transitive extension of all local depend annotations has to be acyclic to make the assumed abstraction a correct one. As demonstrated in Figure 2, the depend relation restricts the valid embedding of a component. This way, an explicit and complete synchronization specification can be avoided.

Component-based systems require that all specific contractual relations of a component are explicitly specified. Besides the usually obvious case of used contracts also assumed standard libraries are part of the contractual relations of a component. However, the explicit treatment of them is rather circumstantial and therefore an implicit *context* is usually assumed. E.g., in the case of EJB [9] components the guaranteed context is fully

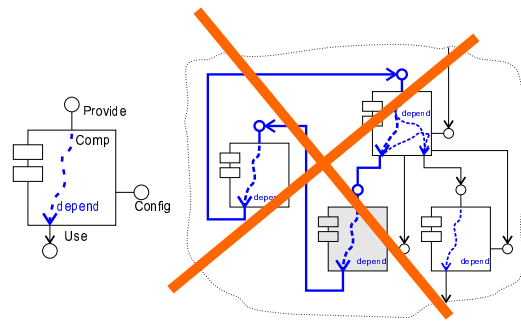


Figure 2. Embedding and depend relation

specified. We therefore extend the TYCS approach with explicit modeling of context capabilities. To handle even cross-cutting aspects however a more generic interference with the context is required. Such a generic contract for cross-cutting aspect integration has to address what interception-points are supported by the component. The component context can further use this generic interface to weave environment specific variations for cross-cutting aspects with a given component. While for a virtual machine based language like Java the integration of such generic interception points seems feasible, the support for components as executable programs is nearly impossible. Note, that the supported interception point have to be further restricted to rather technical cases such as method calls for classes.

3. Example

We use the example of a flat directory service which provides callbacks to an observer of [15, p. 52] to explain our solution for callbacks and intermediate states for component-based systems. In [15], an initial `DirectoryDisplay` client using the `Dir` contract to implement an observer which updates its display for the directory contents each time an entry is added or deleted is considered. There, also a second version `DirectoryDisplay2` is presented, which additionally checks for each notification that the entries do not have the name "Untitled" and directly erases them otherwise.

For an also presented `Dir` implementation that works well with the `DirectoryDisplay` client it is shown that the straightforward implementation using a `notify` callback to erase wrong named entries of the second version `DirectoryDisplay2` results in a stack overflow due to an unexpected infinite recursion. Our notion for components ensured that the observed infinite recursion can not occur even when no detailed knowledge about the concrete implementation exists. The TYCS approach instead suggests to specify an abstract depend relation. If no depend relation between a provided and used contract of a single component exists, synchronizing calls to that contract are forbidden. We consider the original example to demonstrate how to apply this concept using the contract `NotifyingDir`.

In Figure 3 two cases covering the original example

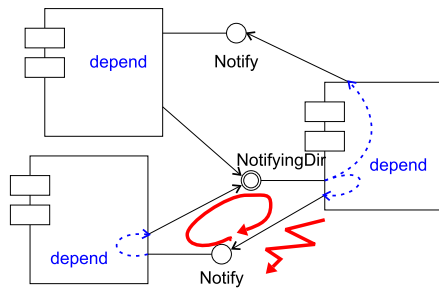


Figure 3. Usage of a Dir contract

using the `NotifyingDir` contract are presented. In the original example the first implementation `DirectoryDisplay` does not access the `Dir` contract during the processing of a `notify` request. Thus, no dependency between the provided `Notify` contract and the used `Dir` contract is given and the upper case of Figure 3 is correct. In contrast does the `DirectoryDisplay2` client implementation access the `Dir` contract in the case of an added name "Untitled". The provided `Notify` contract thus depends on the used `Dir` contract and the lower case of Figure 3 which indicates a possible error is the only one which covers this implementation. The cycle in the resulting combined `depend` and usage relation indicates this possible error. Thus the TYCS notion for components excludes that a composition does not reveal the possible error.

The `DirectoryDisplay2` implementation is intended to ensure a specific rule for the set of stored file names of the directory: no file is named "Untitled". This behavior is modeled more appropriate by extending the concept of a directory and allowing to append special rules to a directory service, e.g., special naming conventions or forbidden characters, that will be checked for every `add` request. This concept does include the scenario with notification and direct remove of the original example but provides a concrete architecture as well as a more systematic way to extend a directory service in a restricted manner. This way a very flexible extension mechanism can be supported at run-time using a set of so-called guards. For each directory, a set of guards which can register and unregister ensure that every newly added file does conform to their implemented rules.

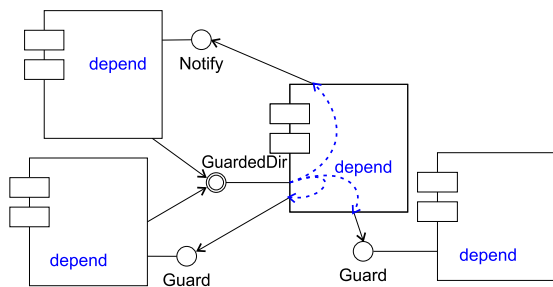


Figure 4. GuardedDir contract

A scenario for the resulting structure is presented in Figure 4. A client with callback notification as well as a client realizing a guard is shown. Note that the guard contract does not depend on any other used contract and thus problems related with callbacks are excluded. Even for the case of a partial view with access restrictions ensuring that every using client has control w.r.t. its added files and their name binding the same extension is feasible.

4. Roles and Separation

Responsibility-driven design of Wirfs-Brock et al. [17], and role-based modeling approaches such as *OOram* of Reenskaug [13] are appropriate design methods for the presented approach. In contrast to data-driven design, the responsibility-driven design attempt to avoid both *centralized* and *overly distributed* designs, by instead attempt to build systems where behavior and data are well distributed and tasks suitably delegated.

Support for behavioral contracts becomes particularly advantageous when a specific design style with emphasis upon separation, for example, component-based design is used. In doing so, it also harmonizes better with methods which support and emphasize modularity. The TYCS contract concept provides a solid ground to establish system interconnections in open systems, but in supporting component-based systems besides the contract notion, the components and their composition require specific handling so as to ensure suitable design.

The supported role-based modeling allow the separation of functional aspects by means of contracts. The different roles a specific subsystem can play within a design can be described by distinct contracts and are therefore separated from each other. During further design activities these distinct contracts can be employed as required while keeping them separated where possible. Thus the overall design complexity is reduced in contrast to a handling which considers the whole class instead of it role specific contract during system construction.

For the directory example we consider that besides the common usage via the `NotifyingDir` contract also administration activities are required. A role-based modeling solution would instead of extending the given contract evaluate whether an additional role is suitable. Installing a guard for a given directory is for example an activity usually restricted to trusted clients. An additional administration role is therefore a useful extension. In Figure 5 such an additional role `AdminDir` is realized using a second contract. Both offered contracts of the component therefore serve the identified different roles and thus separate their clients.

5. Cross-Cutting Aspects

Open systems and runtime binding of services are fundamental requirements bigger distributed systems bring into the game. Temporary cooperation of independent developed components that do not share any or-

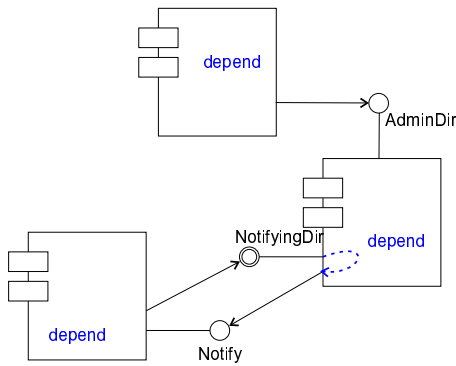


Figure 5. Dir contract and context with name conventions

organizational background are therefore mandatory. Classical class-based designed systems as well as programming language-based approaches for advanced separation by means of weavers will however fail under such conditions. During system design and implementation no fixed boundary or common code basis can be assumed and therefore developed techniques have to be adjusted.

A first fundamental change will be that a system has to operate in multiple possible contexts instead of a fixed static one. Therefore, weaving has to be adjusted accordingly. For predefined interception-points there already exists the technique of runtime service lookup [14] which can be used to do the weaving "on the fly".

The TYCS approach supports efficient runtime contract matching which includes protocol conformance [3]. In combination with explicit context dependencies in form of required context capabilities that are needed to obtain policies or services for well understood orthogonal aspects like tracing, security or failure handling the available technology can provide advanced separation of concerns. However, pre-defined interfaces for service lookup are needed to enable their integration.

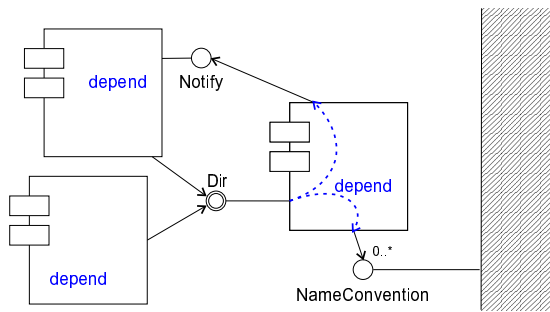


Figure 6. Dir contract and context with name conventions

This option can be exemplified within the directory example assuming a given set of name conventions instead of explicit connected guards. In Figure 6 instead of the explicit Guard contracts the possible empty set

of NameConvention contracts offered by the context are used to exclude that w.r.t. local conventions incorrect names, e.g., the name "Untitled" or invalid characters containing names, can be used.

The need for context capabilities can be further reduced using dynamic overwriting. When required policies are not provided by the context a default solutions operating locally for the single component is used, however, it is often useful to make the context dependencies explicit and demand them as definitive prerequisite for running the component. The propose concept to model the component context rather explicitly and determine the interception-points for the aspects a component should support at design and implementation time.

The presented solutions are however not able to add arbitrary aspect. Instead, only such aspects the component design and implementation is aware of can be supported. Another common approach to address the issue of dynamic adaption and cross-cutting aspects is *configuration* at time of deployment as realized by the EJB [9] component approach. Here, for example, the persistence aspect can be determined when installing a component. Note, however, that again the pre-defined component life-cycle of a EJB component ensures that a well-defined interface exists. Instead of specific modules realizing a particular aspect, the application server permit only to chose from a predefined list of options.

In contrast middleware interceptors [10], e.g., for thread management issues, can be used the realize the commonly used concept to extend a class by adding code when enter and leave one of its method in a generic way. However, the scope of such attempts is restricted to a single executable connected to a specific middleware and therefore does not cover flexible open systems.

While the explicit context contracts can be used to achieve a flexible late-binding or configuration of components, the generic support for interception-points as generalization of the discussed middleware interceptors is proposed by the approach.

When for example a component support to intercept a number of specific events for tracing the in Figure 7 depicted scenario might occur. Note that only in case of an explicit support for a given interception strategy the by the context required support for the Trace contract can be fulfilled.

If a second component want to observe the trace events in a strict synchronous manner the before discussed synchronization effects may also occur. Therefore, when such interference is permitted the context has to publish dependencies right like a usual component. Thus, liveness properties can also be guaranteed for system which support interception points or explicit context contracts.

For components and system modules of different origin and constructed with rather distinct engineering culture the generic solutions such as interceptors as suitable means to realize aspect weaving at runtime also result in some risks. While it is attractive w.r.t. system main-

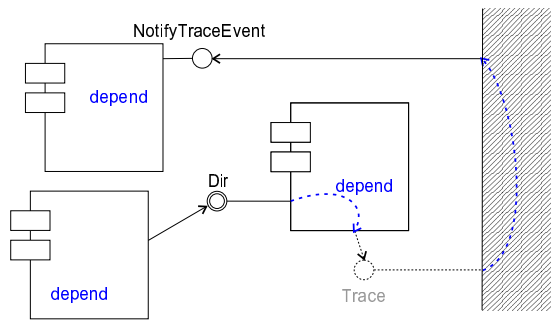


Figure 7. Dir contract and context with interception-point

tenance and code compactness to transfer the benefits of programming language-based advanced separation of concerns concepts to this level, we have to consider also that the risk of unexpected interoperability problems and their impact on system reliability has often more serious consequences in distributed system.

6. Conclusion

Advanced separation of concerns can be one approach to achieve the required technological improvements to rule distributed component-based systems. For the coordination aspect emphasized in the TYCS approach, the described separation of functional aspects via roles is achieved using method signatures and contracts.

For orthogonal aspects the application of explicit context modeling and interception-points have been proposed as solution to extend the TYCS approach. For the case of well-defined interfaces the aspect integration can be realized straight forward. Whether it is appropriate to support the integration of aspects via a generic notion of interception-point the component designer was not aware of however remains an open question.

References

- [1] A. W. Brown and K. C. Wallnau. The Current State of CBSe. *IEEE Software*, 15(5):37–46, 1998.
- [2] H. Giese. Contract-based Component System Design. In J. Ralph H. Sprague, editor, *Thirty-Third Annual Hawaii International Conference on System Sciences (HICSS-33)*, Maui, Hawaii, USA. IEEE Press, Jan. 2000.
- [3] H. Giese. Object Coordination Nets 3.0: Synchronization Behavior Typing for Contracts. TechReport 2/01-I, University Münster, Computer Science, Distributed Systems Group, Feb. 2001.
- [4] H. Giese. *Object-Oriented Design and Architecture of Distributed Systems*. Phdthesis, Westfälischen Wilhelms-Universität Münster, Fachbereich Mathematik und Informatik, Feb. 2001.
- [5] R. Hilliard. Aspects, Concerns, Subjects, Views, ... In *First Workshop on Multi-Dimensional Sepa-*

ration of Concerns in Object-oriented Systems (at OOPSLA '99), 1999.

- [6] E. A. Kendall. Role Model Designs and Implementations with Aspect-oriented Programming. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications, November 1-5, 1999, Denver, Colorado, USA*, pages 353–369, 1999.
- [7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. Techreport, Feb. 1997. XEROX PARC Technical Report, SPL97-008 P9710042.
- [8] S. S. Lam and A. U. Shankar. A theory of interfaces and modules i-Composition theorem. *IEEE Transactions on Software Engineering*, 20(1):336–355, Jan. 1994.
- [9] V. Matena and M. Hapner. *Enterprise JavaBeans™ Specification*. Sun Microsystems, Dec. 1999. Version 1.1.
- [10] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using Interceptors to Enhance CORba. *IEEE Computer*, 32(7):62–66, July 1999.
- [11] Object Management Group. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999. OMG document ad/99-06-08.
- [12] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [13] T. Reenskaug, P. Wold, and O. A. Lehene. *Working with Objects: The OOram Software Engineering Method*. Addison-Wesley/Manning, 1996.
- [14] Sun Microsystems. *Jini Specification*, Jan. 1999. Revision 1.0.
- [15] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [16] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 1999 international conference on Software engineering May 16 - 22, 1999, Los Angeles, CA USA*, pages 107–119, 1999.
- [17] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.