Zaha, J. M.; Albani, A.: Compatibility Test for Coordination Aspects of Software Components. Australien Software Engineering Conference (ASWEC 2006). IEEE Computer Society Press. Sydney, Australia 2006, S.41-48.

Compatibility Test for Coordination Aspects of Software Components

Johannes Maria Zaha
Faculty of Information Technology,
Queensland University of Technology
GPO Box 2434, Brisbane, Australia
j.zaha@qut.edu.au

Antonia Albani
Chair of Business Informatics and Systems
Engineering, University of Augsburg
Universitätsstr. 16, 86135 Augsburg, Germany
antonia.albani@wiwi.uni-augsburg.de

Abstract

Combining third party software components to customer-individual application systems requires first, standardized specification techniques for describing the technical as well as the business-related aspects of the services provided and required by the corresponding software components and second, automated compatibility tests in order to identify components fulfilling demands specified by component requestors. Adequate techniques for the specification of component services are consolidated in a multi-layered specification framework, where formal notations are preferred in order to enable the execution of automated compatibility tests. These tests are a prerequisite for the existence of component markets where third party software components are traded and components that fulfil the specified demands are identified. This paper presents an algorithm for the layer of the specification framework where coordination aspects of a software component are described. On this layer an extension of the Object Constraint Language (OCL) by temporal operators is used to specify the succession relationships between the services of related software components. Thereby the connections to other layers are tagged and existing tests are integrated.

1. Introduction

The idea of application systems made up from prefabricated software components that could be exchanged via component markets, has been traced at least since the publication of McIllroy in 1968 [1]. Compositional reuse of software is a technique to combine the advantages of both standard software and individually programmed software by a plug-and-play-like reuse of black-box components, which are traded on component markets. In the last decades, techniques that enable the reuse of software code contributing to the realization of the idea have been developed. Such techniques are e.g. code and design scavenging [2] or generative programming [3]. In addition to such techniques developers and users need to be able to express the characteristics of a software component in a standardized way, in order to enable the reuse of black-box software components. A standard specification of software components together with automated compatibility tests for all specification aspects allow companies to search for suitable components in repositories and to integrate them in an application system with small effort.

In this paper a test for coordination aspects of business components is provided. In section 2 the definition of the terms software and business component is given and the specification techniques for describing all relevant aspects of a business component – on which the compatibility tests are based on – is introduced. In section 3 the dependencies to other specification artefacts is illustrated as far as this is necessary for the implementation of the automated compatibility tests. Before presenting the test algorithm and an example of use, logical principles, which are applied while executing the algorithm are depicted. Finally, conclusions are drawn and an outlook on future work is given.

2. Business Components

According to [4], a software component consists of different (software-) artefacts. It is reusable, self-contained and marketable, provides services through well-defined interfaces, hides its implementation and can be deployed in configurations unknown at the time of development. A business component is a software component that implements a certain set of services out of a given business domain.

To trade business components on component markets, it is necessary to standardize and to specify them. Specification becomes more important with respect to third party composition of business components, since the specification might be the only available support for a composer who combines business components from different vendors to an individual application system.

2.1. Specification of Business Components

To specify business components in a standardized way, a specification framework has been proposed by [5]. This specification framework (cf. figure 1) considers the description of technical as well as business related aspects and builds the basis for the implementation of automated compatibility tests.

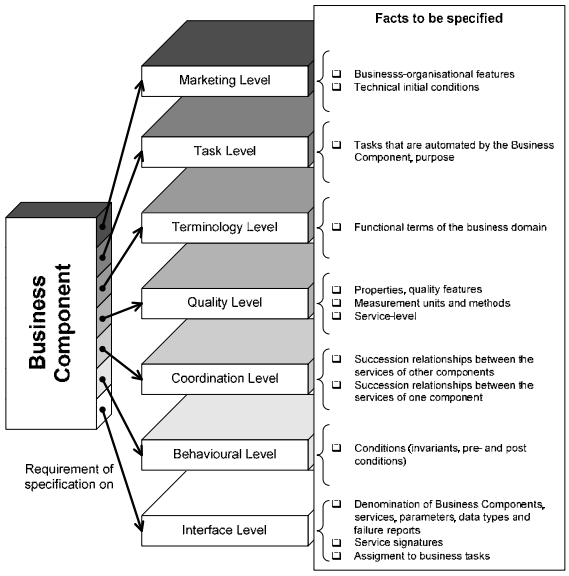

Figure 1. Multi-layer specification framework [5]

In this multi-layer specification framework standardized techniques for the specification of business components on the different levels of abstraction have been chosen using e.g. the Interface Definition Language (IDL) [6] on interface level, the Object Constraint Language (OCL) [7] on behavioural level or the Restructured Business Language [8] on task level. With the use of this specification framework it is possible to describe the complete business logic of and also the quality aspects related to a business component. The framework therefore does not only build the basis for automated compatibility tests, as described in this paper, but also for the implementation of the concept of design by contract [10], which is based on the concept of software contracts as introduced by Meyer [9] with his programming language Eiffel.

While building application systems based on business components, the design phase should end with a detailed specification of all components needed by using the standardized specification techniques just introduced. With this concept it is possible to decrease the effort of building application systems by reusing existing components. In order to automatically find required components it is necessary to use algorithms for automated compatibility tests on all of the layers of the specification framework mentioned above. In this paper the algorithm for compatibility test on coordination level, where a temporal logic [11] has been chosen as formal specification language, is introduced and described.

2.2. Specification on Coordination Level

The purpose of the specification on coordination level is to provide relevant information on how a business component can be integrated into a component based application system from a process point of view. Hence, the specification refers to conditions that have an economic, objective-logical relation to other components or to other services within the business component itself.

According to [4], on coordination level succession relationships between services and synchronization requirements are specified. The specification artefacts on coordination level may refer in the same manner to services offered by the business component itself as they may refer to services required from other components. If there is no need to specify exactly which component provides the services required, or the component is not known, the service will be addressed to a component called external component. The following unary and binary temporal operators can be used for the specification on coordination level:

- *sometime_past X*
- *always_past X*
- *sometime X*
- *always X*
- *initially X*
- *X until Y*
- *X before Y*
- *X sometime_since_last Y*
- *X always_since_last Y*

The expressions X and Y can hold a logic expression that is subject to changes during its lifecycle. E.g. X could relate to the fact that at a

specific point in time the amount of money m stored on an account a is greater than or equal to a given security amount s. That means that X=(m,a,s: m(a)>=s(a)). Looking at the temporal expression always_past X – which means that from the perspective of the actual point in time X has always been valid in the past – with the given X, the expression is only true if in the past the amount on the account has always been greater than or equal to the security amount. For further details and exact semantics of the temporal operators, refer to [11].

In addition the operators before and after are introduced. The operand of both operators is a request of another service including calling parameters. Thus, these operators have to be distinguished from the temporal operators introduced above, since their operands are services and not logical expressions.

- The expression *before(service1(par1,par2))* is true in exactly that moment, when the service *service1* is being requested with the parameters *par1* and *par2*.
- The expression *after(service1(par1,par2))* is true in exactly that moment, when the execution of the *service1* with the parameters *par1*, *par2* has been successfully terminated.

Expressions such as before(...) and after(...) can be used as operands in conjunction with other temporal operators. By means of this extension, the moment of service requests can be expressed syntactically more precise with OCL expressions. Furthermore, it is possible to specify conditions relating to different states during processing of a transaction. To complete the key words that can be used for specification, the operators or, and and not can be used for connecting valid expressions.

The usability of this notation for specification of business components has been proved in numerous case studies – among them one very extensive in the field of Strategic Supply Network Development [12]. These case studies indicate that it is not necessary to use notation artefacts provided by the OCL, but that it is sufficient to use the temporal operators presented above with inserting services and parameters at the respective placeholders. Hence, in this paper a restriction to the temporal operators is made and no introduction to the grammar of the OCL is given.

3. The Algorithm

There are two ways in testing the compatibility of software components, namely by following a formal or a heuristic approach.

For proving the compatibility of two temporal-logic expressions in a formal way existing work in the fields of type systems and compatibility of protocols (cf. [13], [14], [15], [16]) needs to be taken into account building the basis for the formal tests. The operators introduced above would therefore need to be transformed into a finite state machine. But since [17] showed in a formal demonstration that this is not possible, a heuristic approach has been pursued in the work presented and will be described in detail next.

To provide an algorithm for the test of two specification artefacts on coordination level, first of all a definition of compatibility must be given. In this context the dependency between specification on coordination and interface level is described and rules for transforming specification artefacts in logical equivalent expressions are given. Subsequently, the procedure of the algorithm is presented concluding with an example for the execution of the algorithm.

3.1. Definition of Compatibility

To test the compatibility of two specification artefacts on coordination level, the logical equivalence of two temporal logic expressions has to be proved. Thereby the syntactical structure of two expressions is decisive. Since the specification artefacts on coordination level represent a temporal extension of propositional logic, two specification artefacts are equivalent if they represent the same truth function. With the aid of truth tables the logical equivalence of expressions in propositional logic can be verified. This holds for the temporal operators, but not for the services and their parameters, which are defined on interface level, since these are the operands of a temporal-logic expression. Thus, the equivalence of services and their parameters is not within the scope of this paper. The definition of compatibility for these constructs and an automated test on equivalence has been presented in [18].

3.1.1. Dependency on Interface Level

The compatibility algorithm on interface level [18], which is also used for testing parts of the specification on coordination level as just mentioned, compares IDL declarations and tries to construct an unequivocal mapping of these declarations – generating adapters if necessary. To reduce complexity, this compatibility algorithm is divided into sub-algorithms, which can be refined. Among these sub-algorithms there is one for testing the equivalence of services offered by a business component and another to test the equivalence of parameters of those services. These

compatibility algorithms, which are heuristic algorithms for testing syntactical equivalence on interface level, are used during the execution of the test algorithm on coordination level. If the test for temporal artefacts on coordination level is successful, the services and the according parameters – thus the variables of the logical expressions – are passed to the interface level sub-algorithms returning either compatibility (and possibly a set of adapters) or incompatibility. This result is incorporated in the construction of the result of the compatibility test on coordination level.

3.1.2. Logical Rules

The aim of the compatibility algorithm on coordination level is to automatically verify, if two specification artefacts are logical equivalent. This is given, if both of them or any pair of logical equivalent expressions of each of the specifications have the same truth function. These logical equivalent expressions are constructed by applying the logical rules. There are two groups of logical rules: rules for propositional logic [19] and equivalence rules according to Emerson [20] for temporal operators. The rules for expressions in propositional logic state transformations for the operators and, or and not and can be divided in the two rules from De Morgan and the two Distributive Laws [19].

Since realistic specification artefacts are usually strongly nested, the expressions are additionally represented as a tree structure in the following. Thus, the readability is enhanced and the procedure of the algorithm, introduced in section 3.2., which compares two structures and tries to match them, becomes more well-defined.

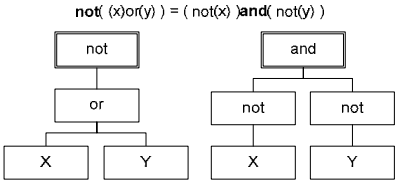

Figure 2. Rule from De Morgan

The first rule from De Morgan is shown in figure 2, whereby the second De Morgan rule can be generated by replacing every or by an and and every and by an or. The same operation leads to the second Distributive Law, if it is applied to the first Distributive Law, displayed in figure 3.

The literals X, Y, and Z represent sub trees that are not specified any further. They can be services with their parameters or complex sub trees. In the first case they need to be passed to the algorithm for the test on interface level, in the second case the use of further logical rules needs to be examined.

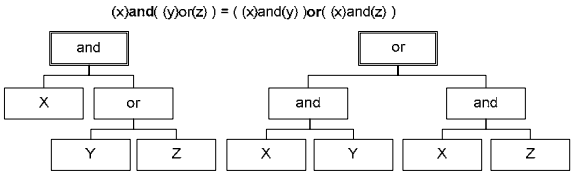

Figure 3. Distributive Law

The rules for temporal operators according to Emerson [20] state equivalence only for few operators. In [20] the existence of operators like sometime_past X or always_past Y is mentioned, but there are no rules defined for transforming these expressions into syntactically equivalent expressions. Therefore there are no rules considered for these two operators and for the operators X sometime_since_last Y and X always_since_last Y.

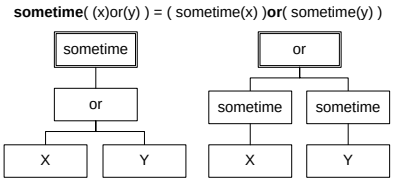

Figure 4. Distributive Law according to Emerson for unary operators

The transformation rules for logical operators in combination with the temporal operators can be divided into Distributive Laws and further Equivalence Laws. Again, they are represented as tree structures in figures 4 to 7.

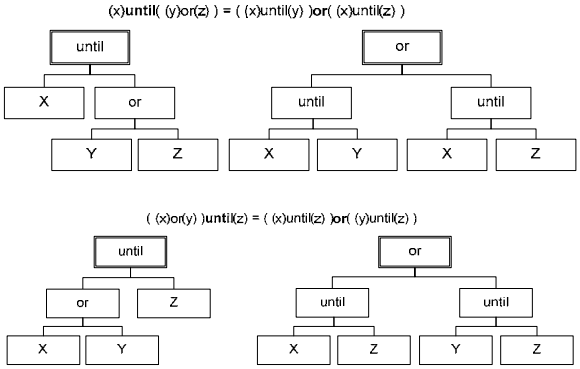

Figure 5. Distributive Laws according to Emerson for binary operators

This Distributive Law according to Emerson displays the distributive character of the operator or (see figure 4). Two more rules can be generated by replacing every sometime by either always or initially. Since the operator sometime_past introduced above is more restrictive than the operator sometime (for a formal demonstration cf. [17]), the algorithm takes every sometime_past in a specification as sometime.

The trees in figure 5 are Distributive Laws for the operator until, once for the disjunction of two variables being the right sub tree of the operator until and once for being the left sub tree. Here again two more rules can be gained by replacing every or by an and.

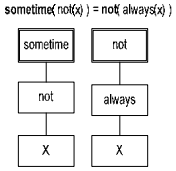

Figure 6. Equivalence Laws according to Emerson for unary operators

The transformation rules depicted in figures 6 and 7 finally determine the substitution of the operator sometime by the operator always and the operator until by the operator before.

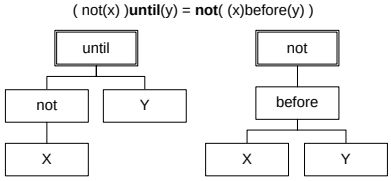

Figure 7. Equivalence Laws according to Emerson for binary operators

The rules introduced above, defining the transformation of a temporal logic expression into a logical equivalent expression, represent the most important rules and are therefore used during execution of the compatibility test. If further rules should be considered, the algorithm could be easily extended.

3.2. Procedure of the Algorithm

As shown in figure 8, the algorithm starts with testing the identity of both trees with respect to the commutative character of the operators or and and, i.e. if there is a differing sequence of two temporal operators connected with or or and, the two operators are switched in order to achieve a match of two trees.

If a match of two trees is possible, the algorithms returns true for indicating the logical equivalence, otherwise equivalence rules that could be executed are searched. In case that there are no equivalence rules available, false is returned for indicating that no logical equivalence could be proven automatically. This result is not a final decision, because only predefined equivalence rules are considered. Although these rules are the most important ones, they constitute only a part of all possible rules. Hence, if the algorithm returns true, the two trees are indeed logically equivalent, but if the algorithm returns false, there is a high probability that the trees are not logical equivalent, but there might exist a logical rule, which has not been considered in the algorithm, that would lead to a positive result. For this reason, every time the algorithm returns false, the part of the tree, where the equivalence could not be proved, is marked such that a human user could examine the problem manually. This mark is likewise used for services that could not be matched by the interface algorithm.

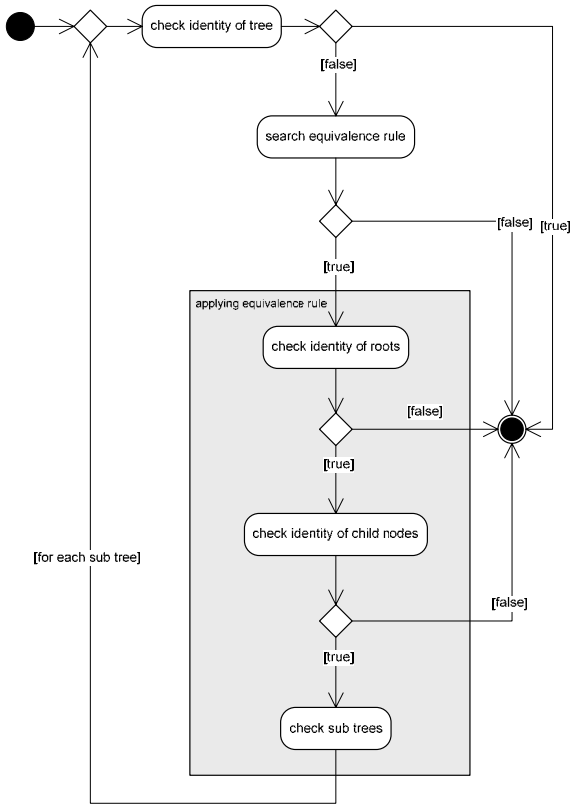

Figure 8. Procedure of the compatibility algorithm

If equivalence rules have been found, they are executed in the following way. First, the root of the

first specification tree has to be identical with one of the roots of the rule tree. The other root of the rule tree has to be identical with the root of the second specification tree. If a match is possible, the algorithm will try to match the direct child as well. After having applied the equivalence rule, the sub trees need to be tested. Thus, the algorithm is recursively called for each sub tree of the specification tree examined.

3.3. Example

The algorithm will be demonstrated on the following example (cf. figure 9), showing two conditions for the service AcceptOrder (order:Order) of two business components BC_A and BC_B. Both conditions determine, that for the execution of the service AcceptOrder with a specific order, it is necessary that the services DeleteInvoice and DelteCustomerData have not been executed for this specific order or, that the service DeleteOrder has been executed for this specific order.

BC_A::AcceptOrder (order:Order)
(sometime_past(not((DeleteInvoice(order))or
(DeleteCustomerData(order)))))
or (sometime_past(DeleteOrder(order)))

BC_B:: AcceptOrder (order:Order)
sometime_past(((not(DeleteInvoice(order)))and
(not(DeleteCustomerData(order))))or
(DeleteOrder(order)))

Figure 9. Example specification

Since the matching of the signatures of the services is tested with the algorithm introduced in [18], they are replaced by the labels ServiceX, ServiceY and ServiceZ in the following. The two conditions can be represented as trees as depicted in figure 10:

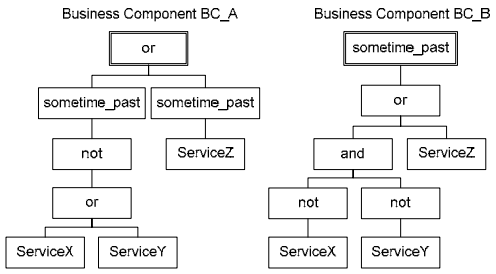

Figure 10. Tree of example specification

First, the algorithm tests if the two trees are equivalent without applying any transformation. Since the labels or and sometime_past are different, an equivalence rule is needed that has these two labels as combination of tree roots in order to examine the two tree structures on logical equivalence. In this example the algorithm identifies the first Distributive Law according to Emerson (cf. figure 4) as equivalence rule that could be applied. Subsequently, for each sub tree in the equivalence rule the child nodes and their connections have to match the sub trees in the assigned tree of the specification. In the example the sub trees of the specification tree of business component BC_A have to match the sub trees of the second tree of the Distributive Law (cf. figure 11).

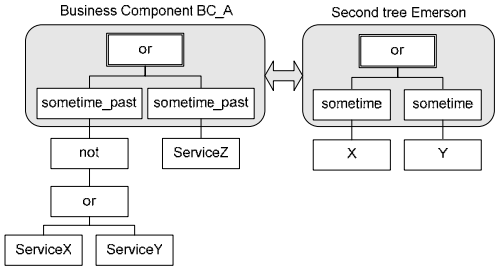

Figure 11. Matching of specification trees of *BC_A*

The sub trees of the specification tree of business component BC_B have to match the sub trees of the first tree of the Distributive Law (cf. figure 12).

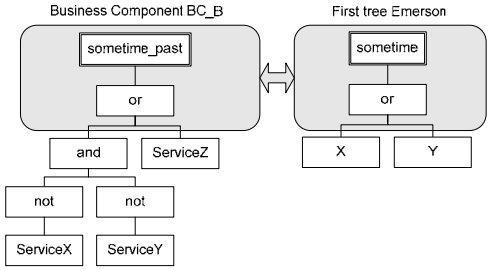

Figure 12. Matching of specification trees of *BC_B*

Since the algorithm could match the connections of the sub trees, the algorithm is started recursively by testing the equivalence of the sub tree of BC_A which corresponds to the sub tree with label X of the second tree of the Distributive Law and the sub tree of BC_B which corresponds to the sub tree with label X of the first tree of the Distributive Law. The matching of the sub trees corresponding to the sub trees labelled with Y is done respectively, whereby the algorithm is able to match the sub trees without applying any transformation.

For the sub trees corresponding to the sub trees labelled with X, the algorithm is started recursively. In this pass an equivalence rule in order to examine the syntactical equivalence of the sub trees has to be

searched, since the roots of the sub trees not and and are not identical. The equivalence rule that has this combination of root nodes is the first rule of De Morgan (cf. figure 2). There the algorithm is able to match the root nodes and the child nodes of the sub tree BC_A to the first tree of the tree according to De Morgan (cf. figure 13) and the root nodes and the child nodes of the sub tree BC_B to the second tree of the tree according to De Morgan (cf. figure 14).

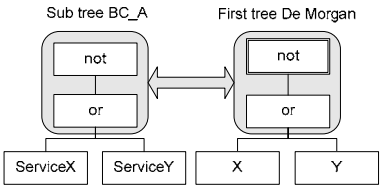

Figure 13. Matching of specification sub tree of *BC_A*

Now the algorithm is started two more times for the sub trees corresponding to the leaf nodes of the trees of the equivalence rules. In this pass of the algorithm a matching can be achieved without applying any transformation, because only services are left in the sub trees.

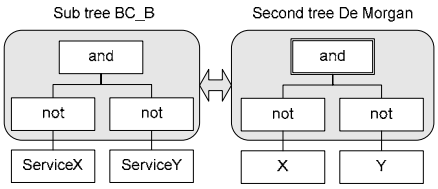

Figure 14. Matching of specification sub tree of *BC_B*

For returning the final result of the example, the algorithm would hand over the services ServiceX, ServiceY and ServiceZ to the algorithm for the test on interface level. Presuming that the services and their parameters of both specifications could be matched, the algorithm would return as final result, that the conditions for the execution of the service AcceptOrder (order:Order) offered by the business component BC_A and BC_B are logical equivalent and that interface specifications of the according services could be matched too.

4. Summary and Outlook

The algorithm introduced in this paper tests the equivalence of two specifications on coordination level, describing the succession relationships of services offered by business components. Thereby, a standardized specification technique is considered, which is adapted according to experiences made in realistic case studies. This algorithm has been implemented and integrated in a tool for specifying business components according to [5], whereby the combination with an algorithm for the test of interfaces [18] has been realized.

Moreover tests on compatibility of business-related aspects of software components as stated in [21] have been developed and integrated in the tool for specifying software components. Thus, the following layers according to [5] can be tested automatically when searching existing software components: task level, terminology level, behavioural level, and interface level. These tests as well as their integration in the implemented tool are presented in [22]. Thus an important prerequisite for establishing component markets on which software components are traded and software applications can be built with small effort combining the advantages of individually programmed software and standard software has been fulfilled.

References

[1] McIlroy, M.D. Mass Produced Software Components. In: Software Engineering: Report on a Conference by the NATO Science Committee. 1968. Brussels: NATO Scientific Affairs Division.

[2] Sametinger, J., Software engineering with reusable components. 1997; Berlin, New York: Springer. xvi, p. 272.

[3] Czarnecki, K. and U. Eisenecker, Generative programming: methods, tools, and applications. 2000, Boston: Addison Wesley. xxvi, p. 832.

[4] K. J. Fellner and K. Turowski, "Classification Framework for Business Components," Proceedings of the 33rd Annual Hawaii International Conference On System Sciences, 2000.

[5] Turowski, K., ed. Standardized Specification of usiness Components. 2002, Gesellschaft für Informatik, Working Group WI-kobAS (5.10.3) - Component Oriented Business Application Systems: Augsburg.

[6] OMG, ed. The Common Object Request Broker: Architecture and Specification: Version 2.5. 2001, Framingham.

[7] OMG, ed. Unified Modeling Language Specification: Version 1.4. 2001, Needham.

[8] Ortner, E. Methodenneutraler Fachentwurf: Zu den Grundlagen einer anwendungsorientierten Informatik. 1997, Stuttgart.

[9] Meyer, B., Object-Oriented Software Construction. 1988, Englewood Cliffs.

[10] Meyer, B., Applying "Design by Contract". IEEE Computer, 1992. 25(10), pp. 40-51.

[11] Conrad, S.; Turowski, K.: Vereinheitlichung der Spezifikation von Fachkomponenten auf der Basis eines Notationsstandards. In: J. Ebert; U. Frank (Hrsg.): Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik: Beiträge des Workshops "Modellierung 2000" St Goar, 5.-7. April 2000. St. Goar 2000, pp. 179-194.

[12] Albani, A.; Bazijanec, B.; Turowski, K.; Winnewisser, C.: Component Framework for Strategic Supply Network Development. In: Benczúr, A.; Demetrovics, J.; Gottlob, G. (Hrsg.): 8th East European Conference on Advances in Databases and Information Systems (ADBIS-04), 22 - 25 September 2004, LNCS 3255. Budapest, Hungary 2004, pp.67-82.

[13] Liskov, B. H.; Wing, J. M.: Behavioural subtyping using invariants and constraints. ACM Transactions on Programming Languages and Systems (TOPLAS), 1994. 16 (6), pp. 1811 - 1841.

[14] Yellin, D. M., Strom, R. E.: Protocol Specifications and Component Adaptors, ACM Transactions on Programming Languages and Systems, 19 (1997), pp. 292-333.

[15] Bussler, C.: Public Process Inheritance for Business-to-Business Integration, Technologies for E-Services, Third International Workshop, TES 2002, pp. 19-28.

[16] Farias, A., Südholt, M.: On components with explicit protocols satisfying a notion of correctness by construction, DOA/CoopIS/ODBASE 2002 Confederated International Conferences, 2002, pp. 995-1012.

[17] Saake, G.: Objektorientierte Spezifikation von Informationssystemen. Teubner-Verlag, Stuttgart, 1993.

[18] Zaha, J. M.; Geisenberger, M.; Groth, M.: Compatibility Test and Adapter Generation for Interfaces of Software Components. 1st International Conference on Distributed Computing & Internet Technology (ICDCIT 2004), LNCS. Bhubaneswar, India, 2004.

[19] Bronstein, I.N.; Semendjajew, K.A. et al : Taschenbuch der Mathematik. 4. Aufl., Verlag Harri Deutsch, Frankfurt am Main, Thun, 1999, pp. 284f.

[20] Emerson, E. A.: Temporal and Modular Logic. Computer Sciences Department, University of Texas at Austin, Texas, 1995, pp. 6-10.

[21] Zaha, J. M.: Automated compatibility tests for business related aspects of software Components. In: Phd Symposium in conjunction with the OTM 2004 (On the Move Federated Conferences), LNCS. Larnaca, Cyprus, 2004, pp. 834-841.

[22] Zaha, J. M.: Automatisierte Kompatibilitätstests für fachliche Software-Komponenten. WiKu-Verlag, Berlin, 2005.